A Load Balancing Framework for Distributed and Parallel Applications

by

Syed Nisar Ul Haq

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January, 1996

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A Load Balancing Framework for Distributed and Parallel Applications

BY

SYED NISAR UL HAQ

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE

January 1996

A LOAD BALANCING FRAMEWORK FOR DISTRIBUTED AND PARALLEL APPLICATIONS

Syed Nisar ul Haq

Information and Computer Science

SHABAAN, 1416H.
JANUARY, 1996G.

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **SYED NISAR UL HAQ** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER SCIENCE.**

THESIS COMMITTEE

Dr. M. Bozyigit (Chairman)

Dr. S. Ghanta (Member)

Dr. M. S. T. Benten (Member)

Department Chairman

Dean, College of Graduate Studies

Date

Dedicated To

Ammi, Daddy, Saleha, Aijaz, Fazal and Farman

Acknowledgements

(Allah) Most Gracious! It is He Who has taught the Qur'an. He has created man: He has taught him speech (and intelligence). The sun and the moon follow courses (exactly) computed; And the herbs and the trees - both (alike) bow in adoration. And the Firmament has He raised high, and He has set up the Balance (of Justice), In order that ye may not transgress (due) balance. So establish weight with justice and fall not short in the balance. It is He Who has spread out the earth for (His) creatures: Therein is fruit and date-palms, producing spathes (enclosing dates); Also corn, with (its) leaves and stalk for fodder, and sweet-smelling plants. Then which of the favours of your Lord will ye deny? (Al-Qur'an: (Surah Ar-Rahman) 55:001-013)

All Praise is for Allah. The Lord of the Worlds, The Beneficent, The Merciful. Peace and Mercy be upon His Chosen Apostle. I thank Allah the Gracious and the Almighty for having brought me to this stage in my career. I thank Him for the favours which He has bestowed on me. I cannot and do not have words to enumerate them and thank Him. I ask Him not to include me in them who deny His favours and always lead me on the righteous path. Ameen.

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research.

First and foremost, I take this opportunity to express my modest thanks to my parents without whose blessings, support, encouragement and motivation, I wouldn't

have been what I am. I also thank my dearest sister and brothers for the great love and constant support. I thank the Compassionate for blessing me with such a caring and affectionate family.

I express my deep regard and sincere thanks to the chairman of my thesis committee, Dr. Muslim Bozyigit, for his guidance and advice. I thank him for introducing me to the field of Distributed Systems, and sharing with me his expertise. He was always by my side throughout this work, even at odd hours and on weekends. His suggestions, constructive criticism and constant encouragement are a major force behind this work as it stands now.

My sincere thanks are due to the members of my thesis committee Dr. Ghanta and Dr. Benten for their invaluable suggestions and constant support.

Thanks are due to Dr. Mulhem, the Chairman ICS Department, for his constant support and for allowing me to use departmental facilities and resources.

It is very difficult to stay away from home for a long period of time for as strenuous a mission as graduate studies unless there are great *colleagues* and *friends*. I thank them all, some of whose names I might be missing, Naseer, Ather, Kaleem, Hadi, Jaleel, Sajjad, Feroze, Raheem, Azhar, Dr. Sukairi; Messrs Jaweed, Suhaib, Asjad, Husni, Garout, Said, Shahid, Zeidan, Abdullah and Purshu.

Contents

List of Figures

List of Tables

Abstract

Name:	**Syed Nisar ul Haq**
Title:	**A Load Balancing Framework for Distributed and Parallel Applications**
Major Field:	**Computer Science and Engineering**
Date of Degree:	**Shabaan, 1416H. (January, 1996G.)**

This work is on the design and implementation of a general Load Balancing Framework (*LBFW*), for executing distributed and parallel applications over a network of workstations. *LBFW* comprises of a hierarchy of modules. Interface between various components of *LBFW* are provided in the form of protocols. A user interaction library is provided to server as an application programming interface to *LBFW*. The *LBFW* library calls can be issued from any *C* program for using to run distributed and parallel applications. *LBFW* uses a load balancing heuristic to affect task placement. The tasks are assigned to the workstations that complete them in shortest possible time, based on a computed load balancing metric for each task. The metric includes task execution time, inter-task communication, run-time workstation load, and other workstation characteristics. The execution times are refined by recording the task execution history.

The implementation of *LBFW* is done using the *PVM* distributed programming tool. To study the performance of *LBFW*, extensive experiments have been carried out using a "*Generic*" distributed and parallel application. The results have shown merits of using *LBFW* in real life situation.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
Shabaan 1416H. (January 1996G.)

ملخص

الاسم: سيد نصار الحق

العنـــوان: نظام لموازنة أحمال التطبيقات الموزعة والمتوازية

التخصص الرئيسي: علوم الحاسب والمعلومات

تاريخ الدرجـــــــة: شعبان ١٤١٦ – كانون ثاني (يناير) ١٩٩٦

يعرض هذا البحث تصميم وبناء نظام عام لموازنة أحمال التطبيقات الموزعة والمتوازية على محطات العمل. ويتألف هذا النظام من سلسلة من الوحدات المترابطة عن طريق بروتوكولات خاصة. ويوفر النظام مكتبة للمستخدم تعمل كواجهة اتصال للبرمجة التطبيقية. ويمكن استدعاء دوال المكتبة من أي برنامج بلغة "سي" لاستخدام وتنفيذ التطبيقات المتوازية والموزعة. ويستخدم النظام خوارزمية موازنة أحمال تقريبية للتأثير على توزيع الأعمال. وتوزع الأعمال على محطات العمل حتى تنفذ بأقصر وقت اعتمادا على مقياس يتم حسابه لموازنة الأحمال. ويشمل المقياس مدة تنفيذ العمل، ووقت الاتصال بين الأعمال، وحمل محطة العمل أثناء التنفيذ، إضافة إلى بعض الخصائص الأخرى لمحطات العمل. ويتم تحسين وقت التنفيذ بتسجيل كيفية تنفيذ الأعمال سابقا. ويمكن دمج أي خوارزمية تقريبية لموازنة الأحمال مع هذا النظام.

وقد تم بناء النظام باستخدام أداة البرمجة الموزعة والمسماة PVM "آلة التوازي الافتراضية". ولدراسة فعالية النظام فقد تم إجراء تجارب مكثفة باستخدام تطبيق عام يعمل بالبرمجة الموزعة والمتوازية. وقد كانت النتائج إيجابية تشجع على استخدام النظام في أوضاع حقيقية.

ماجستير في العلوم

علوم الحاسب والمعلومات

جامعة الملك فهد للبترول والمعادن

الظهران – المملكة العربية السعودية

شعبان ١٤١٦ – كانون ثاني (يناير) ١٩٩٦

Chapter 1

Introduction

Over the last four decades dramatic increases in computing speed were achieved. This was due to use of inherently faster electronic devices, fuelled by the tremendous development in the realm of *VLSI*. Unfortunately, this trend is about to come to an end. Assume that a device performs 10^{12} operations per second. Then it takes longer for a signal, which is determined by the speed of light, to travel between two such devices one-half of a millimeter apart than it takes for either of them to process it. Again, by the laws of physics, the reduction in the distance between electronic devices is not possible after certain point because this may lead to interference. Thus the only way around, as it appears, is through *parallel computation.* Much has been done in this realm both in the context of hardware and software. Desktop computing in the form of personal computers and workstations, coupled with the advances in communication technology have altered the state of art in computing. Computers are now connected by a communication network in order to share data, resources and provide communication between different computers.

Thus arose a concept of a *Distributed Computing System(DCS)*, which can be

defined as a collection of loosely coupled computers connected by a communication network, allowing the development of distributed applications. There are many examples applications *DCS*. Service based distributed applications include:

- Airline/Railways reservation systems
- Distributed database systems
- Network file systems
- Electronic Mail and File transfer

DCS offer significant computing power to handle complex problems. Parallelism is inherent in *DCS* due the multiplicity of interconnected processing elements *(hosts)*. Thus many research problems which have challenged scientists and engineers, called the *Grand Challenges of the 90s* are now attempted to be solved using *DCS* based parallel computing. Some of these problems are:

- Numerical Weather Prediction,
- Oceanography,
- Socio-Economic and Government use,
- Engineering
  - Finite Element Analysis,
  - Computational Aerodynamics,
  - Plasma Physics etc.
- Artificial Intelligence

- Image, Speech and Pattern processing,
- Computer Vision,
- Robotics etc.

The decomposition of a potentially parallelizable problem into several independent sub-problems and shared execution by a set of processors can decrease the overall computation time. Thus, there is an increase in the speedup and efficiency of the application. This is achieved by utilizing the existing, redundant computing power. The availability of high speed communication with *DCSs* makes them a viable option for parallel applications. Another advantage with *DCS* is that the need for special parallel computers is minimized and in some cases eliminated. Moreover, the *DCS* still fully supports the general purpose usage of other applications.

An issue with *DCS* is that not all problems are suitable to be solved efficiently using this approach. Most of the time they have to be reformatted so that properties of the problem are exploited to enhance parallelism. This also does not guarantee better performance. To ensure it, application characteristics as well as the system characteristics must also be considered while parallelizing and executing it. This has been the focal point of most of the research in the area of distributed computing.

Another important consideration is that its performance characteristics vary temporally. This is primarily because of its *loosely coupled* nature. Each computer in the network is utilized by, typically, a single person. Therefore, the way it might be used is a question too difficult to be answered. Apart from this, another possible scenario is that some of the machines may not be functional.

1.1 Classification of Parallel Architectures

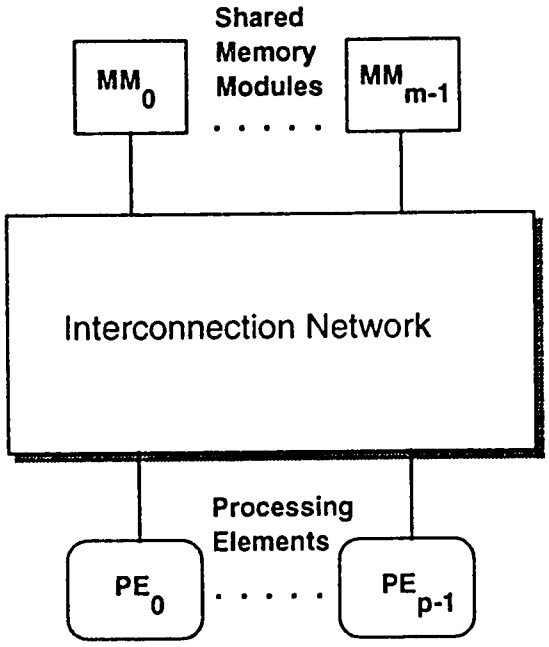
Parallel architectures are broadly classified as being *tightly* or *loosely coupled* [1]. Processors in tightly coupled parallel systems communicate through shared memory (ex. Alliant FX8, BBN Butterfly, Encore Multimax etc.)[1]. Processors in loosely coupled parallel systems communicate by exchanging messages (ex. Intel iPSC, NCUBE-10 and a workstation based distributed system)[1].

1.1.1 Tightly Coupled Parallel Systems

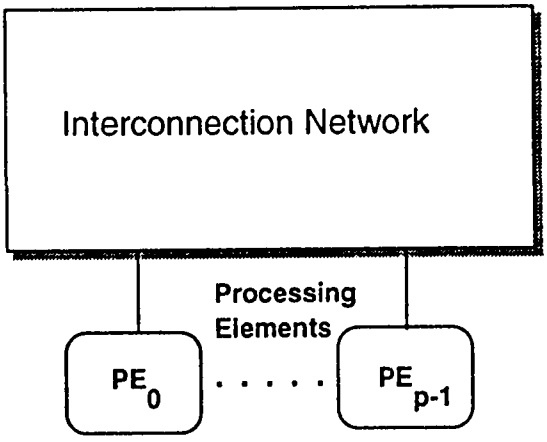
Systems of this category are basically shared memory multiprocessors with an interconnection network, Figure 1.1a, which connects P processors to M shared memory modules. As the number of processors increase, the interconnection network becomes the dominant component of the multiprocessor both in terms of cost and performance. The interconnection as such could be of any topology, but it must be of high speed.

1.1.2 Loosely Coupled Parallel Systems

Figure 1.1b shows the general structure of a loosely coupled parallel computing system. Basic difference from a tightly coupled parallel systems is that processors do not share the memory. The processors communicate by sending and receiving messages. Again their performance and scalability is determined by the interconnection network. The simplest such interconnection network is an Ethernet based local area network. Distributed systems having this structure can handle applications with comparatively large granularity with scalability of upto 30 processors [1].



(a) Shared Memory Mutiprocessor



(b) Message Passing Muti-Computer

Figure 1.1: Broad Taxonomy of Parallel Computing Systems

1.2 Motivation and Objectives

1.2.1 Motivation

The aim of this work is to study the suitability of distributed systems in realm of parallel processing and vice-versa. There is an increasing trend towards parallel processing as is clear by the *grand challenges*. Because of the inherent complexity involved in both the hardware and the software parallel computing systems, they are very costly. They are mostly used for special purpose applications[2]. It is very difficult to utilize them for general purpose applications.

On the other hand, contemporary personal computers and the workstations are becoming increasingly more cost effective. The current growth in the computer networking technology has increased the application domain of workstation based computer networks. The applications that were once associated with special purpose parallel systems, can now be run on workstation based networks. With the increase in efficiency and reliability of contemporary networks, the idea of exploring it as parallel computing system has become feasible. Load balancing is an attempt in this direction.

1.2.2 Objectives

- The main objective of this work is the design and development a load-balancing framework (*LBFW*) for distributed and parallel applications on a local area network of workstations. The framework provides interface mechanisms for the user to write parallel applications as a set of tasks and execute them in a load balanced manner.

- Development of various load balancing strategies for *LBFW*.
- Implementation of a sample parallel application on the developed framework.
- Study the performance characteristics of *LBFW* by changing the tunable parameters and compare the results for different load balancing heuristics.
- Identification and definition of the procedure used to customize load balancing algorithms with *LBFW*.

1.2.3 An Integrated Parallel Programming Environment

This work is a sequel of an ambitious workstation based distributed and parallel programming environment. A global picture of the requirements of such an environment are presented in Figure 1.2. Its components are:

1. *Parallelization/Clustering:* This subsystem is responsible for efficient reformatting of the problem into a number of independently executable *tasks* which communicate using message passing.
2. *Scheduler:* The job of this subsystem is to generate a best possible schedule of execution of the application *tasks*.
3. *Mapper:* This module decides as to which of the tasks, given to it by the Scheduler, are to be executed on which host of the *DCS*.
4. *Monitoring Subsystem:* This performs the job of continuously monitoring the load characteristics of the *DCS* which are used by the Mapper.
5. *Dispatcher Subsystem:* This does the job of actually executing a task on a specified host (of the *DCS*).

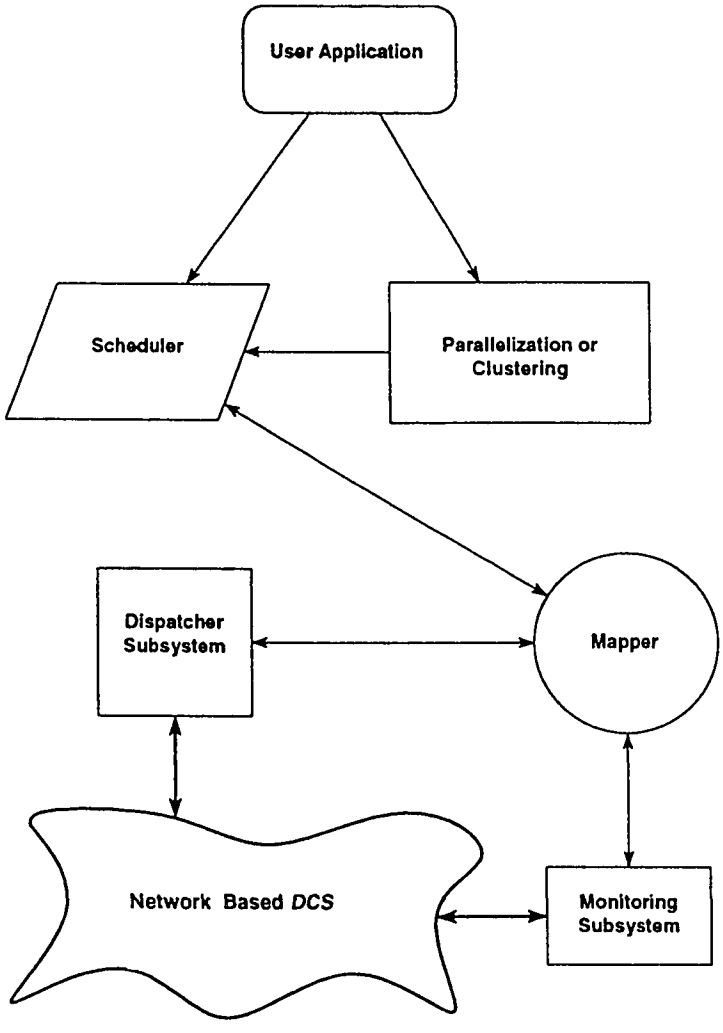


Figure 1.2: Block Diagram of a, cluster of workstations based, integrated parallel programming environment

Each of the phases shown is a research topic in its own right. The job of parallelization is not a simple one. There is no well known method of doing the same, leave alone methods to automate it. Development of parallelizing compilers is one of the active research areas. The problem of scheduling has long been known to be a NP-Hard problem even in the case of uni-processors. But with the multiplicity of computational resources, this becomes even more intractable. Thus one could have a parallelizing compiler for a common programming language, which would breakdown the given program into parallel tasks. The parallel scheduler takes this input and generates a best possible schedule of execution. This schedule is then passed over to the mapper, which maps a given schedule onto the available distributed computing system using the dispatcher subsystem. This work is targeted for designing and realizing a mapper and a task dispatcher subsystem.

1.3 Thesis Layout

Chapter 2 presents a survey of the literature reviewed in the fields of load balancing and network based parallel computing systems. Chapter 3 gives a complete design specification of *LBFW*. All the modules and their interactions are described in detail along with the algorithms used. A load balancing heuristic for *LBFW* is also presented in this chapter. Chapter 4 discusses an implementation of *LBFW* based on *PVM* distributed programming support system. Chapter 5 describes some of the experiments conducted on *LBFW* and some measured performance figures. Chapter 6 concludes the thesis with comments on future work.

Chapter 2

Related Work

2.1 Introduction

Multiplicity of computing resources available in parallel processing systems has fuelled a great amount of research in this field. There are many aspects of parallel computing. Some of them are:

- Developing architecture models for large parallel systems
- Understanding the influence of technology on parallel processing systems design.
- Developing theory and practice for designing parallel algorithms.
- Specification of programming languages tailored for parallel processing.
- Realization of parallel compilers for commonly used programming languages.
- Development of techniques for mapping algorithms and programs onto parallel processing systems and

- Development of methods for evaluating the performance of parallel computing systems.

The main motivation behind our research is related to *how to use widely available DCS efficiently for solving a large range of problems.* One of the ways of approaching this problem is through *load balancing. Load balancing* is a technique for enhancing the utilization of processors and improving the exploitation of parallelism through appropriate distribution of load in a *DCS.*

2.2 Approaches to load Balancing

Many approaches have been taken for studying *load balancing* by the researchers. They are static, dynamic, prediction-based, hybrid, semi-dynamic etc. Almost all of them fall under one the two generic types, static and dynamic.

2.2.1 Static Load Balancing

Distinctive features of Static Approach

The following is a list of characteristic features of most of the static load balancing.

- A priori information about the cost of computing each task and communication between tasks is assumed. It is assumed that this information is available somehow, usually using the characteristics of a set of previous runs of the program and by the knowledge of the algorithm itself.

- The workload is distributed once , as per a plan at the start of the computation.

- No extra overhead is required for load balancing during the execution of the task.

- Problem and system characteristics do not change.

Classes of problem formulations for Static Load Balancing

Items listed below are representative of most of the work done in the area. They are

- Graph Theoretic formulation.
  - The nodes represent modules in a communication graph.
  - The edges represent node communication and the corresponding costs (the weights of the edges). The situation could be modelled by the Max-Flow and Min-Cut problem[3].

- Integer Programming based formulation. The basic idea is to define an objective function subject to a set of constraints.

- Heuristic formulation. The above two approaches are computationally intractable. Hence many heuristics have been proposed which give suboptimal solutions.

2.2.2 A Review of Literature on Static Load Balancing

- Lo [3] investigated the problem of static task assignment in Distributed systems. The work is based on the Graph theoretic approach. The input,*task force*, assumed is a graph consisting of tasks and processor nodes with inter-task edges suitably weighted to represent the communication cost. This is

an extension to the Stone's model and exposes a serious drawback in it, the exclusion of the degree of parallelism or concurrency.

The basic objective function behind the proposed algorithm is the minimization of the total execution and communication costs. An $n - way\ cut$ of this graph is a natural assignment of the processes to the processors. But this is a NP Hard problem. The most important contribution of this paper is a heuristic task assignment method. The trick used is converting the $n - way\ cut$ to a $2 - way\ cut$ and thus reducing the problem to that of Stone's. If this method does not assign all tasks to processors, the rest are assigned in a greedy way so as to minimize the given objective function (this leads to suboptimal assignment). The paper also studies the effect of interference between tasks, if assigned on the same processor.

- Bowen et al [4] deal with the mapping problem of an arbitrary process graph onto an arbitrary processor graph in detail. They present a clustering algorithm which can be used to cluster hierarchically, both the processes and the processors with the objective function of minimization of the communication costs. Then a heuristic mapping algorithm is presented and evaluated. The objective function of this mapping strategy is to minimize the communication cost while keeping the load balancing constraints of minimum and maximum workload assignments in perspective.

  The Clustering Algorithm

  The algorithm presented in this paper is a variant of the Agglomerative type (which considers initially the graph to be set of N clusters of *One* element each. $N - 1$ passes are made through the graph where each pass merges the

most related clusters) as against the Divisive type of algorithms(initially the graph is considered to be single cluster and then successively split until the final clusters are of the desired size). The resultant structure of clusters is logically arranged as a tree.

The paper does not explicitly say it but the same clustering algorithm is applied to the processor graph so that, logically, it has the same hierarchical structure as the process cluster tree. Thus, the mapping process is simplified to the problem of mapping a tree of processes on a logical tree of processors.

The Allocation Algorithm

The allocation algorithm described is recursive in nature with the initial two arguments being the roots of the processor tree and the process tree. The basic idea is to alter the process tree so that the size of the process tree becomes the same as the size of the processor tree, so that the child i of the process sub-tree is assigned to the child i of the processor sub-tree. The method proposed in this paper is radically different from that proposed in [3]. This is another way that researchers look at the problem of static task assignment based load balancing.

Bozyigit et al [5] use an incremental clustering scheme which partitions the parallel applications into clusters of size one to system size (number of processors). It then chooses the clustering that produces the best mapping according to a heuristic objective function. It considers interference costs in computing the parallel application completion time.

2.2.3 Dynamic Load balancing

Characteristic features of Dynamic Approach

The following is a list of salient features of the dynamic approach to load balancing

- A priori information about the cost of computing each task and the cost of communication between tasks is not, throughout the execution period, fixed in contrast to the static load balancing case.
- The concept of *load* is understood as an up to date view of the amount of work being done and/or to be done by the system. It is highly dependant on the system and the run time applications. The system load varies dynamically.
- Load balancing is performed during the running of the application. This results in additional overhead during execution.
- the dynamic nature of load balancing can be visualized under three heads
  - *Job assignment by the user at the process' creation time:* This is a simple solution, but is dynamic in nature. The method puts trust in the intelligence of the user and assumes that the user is going to execute tasks according the load balancing requirements.
  - *Job initial placement by the operating system:* This method relieves the user from the burden of assignment. The operating system or operating environment maintains the load information about the system and whenever a task is presented to it by the user it uses its gained intelligence

from load monitoring and current task information to place the task to maintain the load balancing constraint.

- *Job reassignment using task migration:* This approach is the most general and effective but it increases the complexity involved. The issues involved are
  - * providing the migration facility as such,
  - * deciding when to migrate,
  - * deciding which processor to migrate the task to, and
  - * handling the instability in the migration algorithm eg., processor thrashing.

Thus, effectively, any adaptive load balancing algorithm can have four parts (modules). They are

- *Processor load measurement module:* This module does the job of maintaining an updated load information of the *DCS*. This is termed as the *load state* information.
- *Load state information exchange module:* In order to make a load balancing decision, the load information has to be exchanged between the hosts.
- *Transfer policy module:* A decision has to be made about where to execute or migrate a task before actually executing or migrating for execution at a different host..
- *Cooperation and Location policy:* This module lays down the protocols on which the modules cooperate, share information, and work together.

The paper [6] in [7] gives the basic concepts behind the load balancing strategies with a very brief insight into the possible solutions by going through the gist of some of the work which has been done in the field.

2.2.4 A Review of Literature on Dynamic Load Balancing

2.2.4.1 Load Monitoring

The information about resource loads is a part of any computing system's state and is perhaps the most rapidly changing element. The *load* of a system is an indicator of the amount of *current work* being done by it. The physical quantity or quantities used to extract meaningful load information are called *load indices.* A good load index according to [8] has the following features:

- It reflects the qualitative estimates of the current load on a system
- It allows prediction of the load values in the near future since the response time of a task is affected more by the future than the present.
- It should be relatively stable; i.e., high frequency fluctuations in load must be discounted or ignored.
- It should be easy to compute.

A wide variety of load indices have been used explicitly or implicitly in the literature, in the context of dynamic load balancing. Domenico and Zhou [8] cite many studies that use the ready queue length as the load index.

Kunz [9] sees the problem of load balancing as basically a *Scheduling* problem, determining the host at which a specific task is to be executed such that a system-wide function is optimized. The most important point raised in this paper is of

defining a method (called a *load descriptor*) of finding the current workload of a single host. The paper reported using the following single workload descriptors

- number of tasks in the run queue,
- the size of the free available memory,
- the rate of CPU context switches,
- the rate of system Calls,
- the per minute load Average and,
- the amount of free CPU time

The paper also discusses combination of workload descriptors as a function of weighted combination of the above load descriptors and goes on to prove that it is no better than the best single workload descriptor used independently. The reason is that the overhead in maintaining more than one workload descriptors. This paper gives a good explanation of the concept of *load*, although, it does not explain how to get it. Also no other external interference is assumed during the experimentation.

Mukta [10] bases the discussion on load of a DCS on prolonged gathering of data on the workstation utilization in a university environment. According to this paper, the load of a typical DCS has temporal characteristics. This means that based on the accumulated information over a period of time, one can say that at a particular instant how free or busy the system is. This approach assumes that each job will know its resource requirements, especially CPU. It is a very simplistic approach which does not take network characteristics into account. It might come

in handy for load balancing computation intensive applications on a typical *DCS* whose prolonged usage activity statistics are available.

David Ogle et al present the monitoring subsystem of a general distributed programming system (the Issos System) which collects and analyses monitoring information and makes application dependent monitoring information available to the programmer and to the executing program for adaptation in [11]. The paper describes a very high level information model for representing the program monitoring information, which is based on the ER model. The use of such a model, apart from the other advantages, also promotes language independence of programs being monitored. The other characteristics of the monitor are:

- application dependent monitoring, and
- dynamic distributed data collection and analysis.

This paper gives a broad picture of the design of a distributed monitoring system, the typical functionality needed and the typical interaction between the modules. Since the discussion is about the parts of a complete distributed operating system, it assumes many features usually not available in a typical Unix environment. They include powerful language features for handling distribution, concurrency and sharing. It also throws light on possible methods of tracking load. The issue of tracking and balancing load is tied down to programs (processes) and shared variables.

Goswami et al in [12] base their load monitoring method on a previously developed statistical pattern recognition method to come up with a predicted resource requirements for dynamic load balancing. The prediction is made at the beginning of the processes life, given the identity of the program being executed. The basic

logic is to assign a new process to a CPU with least load. A central node (the scheduler) maintains status of all the nodes. It then

- computes the CPU load for all the nodes,
- identifies the process to be scheduled and feeds it to the predictor,
- Sends the process to the node with smallest CPU load,
- update the status information.

On the receipt of the actual resource utilization information from a completed process

- feeds it to the predictor to update the pattern recognition model,
- modifies the status information

This method is more suitable for *CPU* intensive tasks. It does not take communication into account. The good point is that it displays a learning mechanism which learns about the resource utilization of the hosts.

2.2.4.2 Dynamic Load Balancing Methods

Hac and Jin in [13] discuss two *Sender Initiated(SI)* algorithms. The idea of a *SI* algorithm is that the heavily loaded processors search for the least loaded remote processors for load balancing. Whenever a user submits a job to a host, the *SI* algorithm on this host is invoked to determine whether the job should be processed locally or transferred to a remote host. The workload descriptors used are maximum process queue length and maximum amount of the CPU time of the active processes on each host respectively. The paper assumes a local area network based set of

workstations as the distributed system and assumes that there is no external interference during the experimentation. According to the authors in a large distributed system, a centralized mechanism rather than a distributed monitor approach should be used. Then, the paper tries to prove that the result of load balancing with using the maximum amount of CPU time of active processes on each processor, in the case when the system is loaded with long processes only, does not lead to a balanced distribution of the load for some period of time. However in the longer period of time, the system may become better balanced. This paper does not take the case of a distributed and parallel applications into account. The communication overheads and process are ignored.

As an extension of their work in [13], Hac and Jin present a *Receiver Initiated (RI)* load balancing strategy in [14]. In this strategy the algorithm is based on the idea that the lightly loaded processors search for the heavily loaded ones from which work may be transferred. The system load is balanced in terms of the number of active processes on each host. The major enhancement with reference to [13] is that it considers a *Migration factor* into consideration which is defined as *the ratio of the mean elapsed time of the transfer of a process and the relative file to the response time of the process executed locally.* Thus, if *migrateable factor* is less than or equal to one, then the process is *migrateable* otherwise not. As in [13], distributed and parallel applications are not considered.

Atallah et al report a model for a system that allows the definition of an objective function to be maximized [15]. The class of applications considered require some form of synchronization among the sub-tasks. The following classes of agents are identified: *application agents*, agents that coordinate the execution of an appli-

cation; *decision making agents*, that are involved in resource allocation policies, and *scheduling agents*, that enforce resource allocation policies. The model assumes that the scheduling agents obtain the information regarding the resource requirements of the applications, *somehow*.

Kam and Bozyigit report a load distribution methodology through competition for workstation cluster in [16]. The *DCS* is assumed to be a set of workstation clusters. Akin to typical load balancing algorithms, their approach first merges the tasks of the given application according to the available clusters and maps them based on the concept of the *market price*. The *price* is governed by the demand and supply of traded commodities (processing and communication capacities) and the competition created between the buyers (tasks) and the sellers (workstations) respectively. This assumes the availability of the task communication and execution times of the application and also the characteristics of the *DCS*. Based on this information elaborate algorithms are presented for task merging and mapping based on the concept of market price.

2.2.5 Workstation Networks Based Distributed Computing

Cluster computing or distributed processing on a network of workstations is a rapidly evolving technology. In general, such systems permit a collection of networked machines to be used as a unified general purpose distributed computing resource. While there are commercially available multiprocessor systems, networked collection of workstations can augment their capabilities. The following discussion presents some of the work done, and being done, in this area.

The work of Sunderam and Schmidt[17] investigates the issues in modelling concurrent applications executing in heterogeneous networked environments. Their's is a pragmatic approach rather than analytical. They used the *PVM*[18] as their working environment. They measure the communication time for an Ethernet based network using a variety of well known applications, and fit curves for this as a function of the number of bytes transferred. Then using the models developed and the available analysis of the applications, they measured the predicted times of completion and compare it with the actual values. They report good accuracy in their models based on experiments on different sets of applications. Based on their results they state that main reason of bad performance is the communication network, because:

- the bandwidth of most of the commonly used networks is much lower as compared to traditional multiprocessors,
- there is more contention for the communication medium than the multiprocessors because of
  - traffic due to external loads,
  - paging activity caused directly by concurrent applications in diskless environments and in the case of a Network File Systems, and
  - the inter-task communication and synchronization in a concurrent application,
- heterogeneous computing environments are prone to severe load imbalances, owing to disparities in machine capabilities and external loads.

A framework for partitioning parallel computations in a forest of workstations based distributed computing environment is presented by Wiessman and Grimshaw in [19]. The model involves partitioning the network into clusters on the basis of hardware locality and similarity and then partitioning the application to take the advantages offered by the clusters. The model assumes that the task implementation is provided either by the user or as a result of compilation process. A task executable for each architecture in the network is assumed. The model itself is designed in an object oriented parallel processing system Mentat. The proposed design contemplates the use of *callbacks*, in other words side-effects, in the services provided as a part of the framework to figure out the resource requirements of the tasks. No concrete algorithm for finding the requirements is presented. $SPMD$ based data parallel applications are the ones considered. The idea proposed is to partition data in such a way as to reduce the application completion time.

Peterson and Chamberlain state and validate a performance model for synchronous iterative algorithms executing on a network of workstations in [20]. The class of the applications considered is such that the application contains a number of iterations in which each processor completes its work for the current iteration and joins a barrier synchronization with the rest of the processors. During this period data is communicated between the different processors. The model takes into account dedicated, shared, homogeneous and heterogeneous resources and also the even or uneven distribution of work between them.

The work of Haq et al[21] presents a framework for load balancing distributed and parallel applications on a network of workstations. The issues involved in the interaction between users and the framework and among the various entities of such

a framework are presented from an implementation point of view.

2.2.6 Multiprocessor Load Balancing

Feng and Yeun report experiments on using dynamic load balancing on transputer based *DCS* in [22]. They assume that the tasks are generated at run time and are distributed among the available processors dynamically. They use a *distributed* strategy for load balancing. This is easily possible because of their target architecture. They further assume the presence of kernel level support from the operating system in terms of providing the load status of the processors. It also provides support for sending and receiving data and for synchronization. They report experiments using a Sender, Receiver and mixed strategies for load distribution. Because of the targeted environment, the communication cost is ignored.

LeMair and Reeves in their work [23] present a comprehensive survey of strategies used for dynamic load balancing on multiprocessors. The strategies are classified as follows:

1. *Sender Initiated Diffusion (SID):* This is a highly local approach where a processor makes use of near-neighbor load information to apportion surplus load from heavily loaded processors to under-loaded neighbors in the system. Global balancing is achieved by as tasks from heavily loaded neighborhoods diffuse into lightly loaded area. This scheme is purely asynchronous, i.e., each processor is independent. It has been shown in [24], that for an N processor system with a total system load L unevenly distributed across the system, a diffusion based approach like SID strategy, will eventually cause each processor's load to converge to L/N.

2. *Receiver Initiated Diffusion (RID):* This strategy is converse of SID strategy, where underloaded processors requisition load from heavily loaded neighbors. The balancing is started by any processor whose load falls below a certain threshold. Upon the receipt of a load request, a processor will fulfill the request only upto an amount equal to half of its current load (this is to maintain the stability of the algorithm). Therefore as against the SID scheme, most of the overhead is borne by the underloaded processors.

3. *Hierarchical Balancing Methods (HBM):* This is an asynchronous, global, approach which organizes the system into a hierarchy of subsystems. Load balancing is initiated at lowest levels in the hierarchy with small subsets of processors and ascends to the highest level which encompasses the entire system. These are called *domains.* Each domain has a designated *domain controller*, which has the responsibility of performing load balancing for the processors under it. Such organization finally comes to a stage where there is a central controller processor. Hence this scheme centralizes the balancing process at different levels of the tree with increasing degrees of knowledge at higher levels. The most important advantage of HBM over RID and SID is that migration between distant processors are avoided.

4. *Gradient Model of Load Balancing (GM):* This was originally proposed by Lin and Keller in [25]. This employs a gradient map of *proximities* of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors. The basic concept is that underloaded processors inform other processors in the system of their state, and overloaded processors respond by sending a portion of their load to the the processor having the

smallest *proximity*. The *Proximity* or *distance* of processor j as seen by processor i is the shortest path between them. *Diameter* of a network of N processors is = max $d_{i,j}$ for all i, j in N. All nodes are initialized with a *proximity* of equal to the *diameter* of the system. The *proximity* of a processor is set to 0 if its state becomes lightly loaded. The gradient map of the proximities of underloaded processors in the system serves to route tasks between appropriate processors.

5. *Dimension Exchange Method (DEM):* This is similar to the *GM* strategy. It was conceptually designed for a hypercube system but may be applied to other topologies with suitable modification. In the case of a N processor hypercube, balancing is performed iteratively in $log\, N$ dimensions. All processor pairs in the first dimension, those processors whose addresses differ only in the least significant bit, balance the load between themselves. Next, all processor pairs in the second dimension balance load between themselves. Thus for a $three-cube$, processor pairs $0-1, 2-3, 4-5$ and $6-7$ perform load balancing in the first dimension, $0-2, 1-3, 4-6$ and $5-7$ in the second and finally $0-4, 1-5, 2-6$ and $3-7$ in the third dimension.

Chapter 3

Design of LBFW

3.1 Introduction

The discussion in this chapter is to explain the design of Load Balancing FrameWork (*LBFW*), for providing an operating environment for the execution of distributed and parallel applications in a *load balanced* manner.

The following presents a list of terminology used frequently throughout the thesis.

Task: This is defined as the basic component of an *application.* Tasks are executable programs that run sequentially.

Application: An *application* in the context of *LBFW* is either a *distributed* or a *parallel* application. Applications can be viewed as a collection of independent tasks, that may or may not be communicating, some or all of whom can be executed in parallel.

Traditionally, applications are said to be *parallel* if the degree of coupling inbetween their tasks is high and are referred to as *distributed* in the opposite

case. Degree of coupling in this context means how much the tasks are inter-related in terms of data sharing for solving a given problem.

Formally, an *application* is defined as a *quad tuple*

$$A = \langle T, E, C, N \rangle \tag{3.1}$$

where

T is the application task graph

E is the task execution times

C is the communication volume between different tasks

N is the architecture of the target network on which the application has to run

Network Architecture (N): With the tremendous growth of computer networks, connecting networks has become very important: Hence the rapid growth of the *Internet*. Whatever may be the network architecture, it must be able to cohabit with the Internet easily in order to survive. Therefore, no specific restriction can be made on the type of network architecture suitable for *LBFW*. Thus the whole Internet is visualized as a *DCS*. Thus, a *DCS* can be seen as clusters of workstations in the *Internet*. Let Γ_i denote the set of workstations in a network i. Then in a formal way, *DCS* can be represented as $\Gamma = \{\Gamma_i \mid i \in n\}$.

Granularity: This refers to the number of instructions (computation intensity) of a typical task.

From the basics of parallel processing, we know that there are different levels

of parallelism:

- job Level,
- task Level,
- process Level,
- instruction Level,
- variable Level and
- bit Level.

Job level, task Level and to some extent process level parallelism is what *LBFW* is concerned with. In this context, a multiprocessor can be a part of the network on which *LBFW* runs. Conforming to the objectives of this work, the architecture of interest is a network of workstations. Therefore, task Level parallelism is default parallelism provided by *LBFW*. This discussion is summarized in Figure 3.1. It gives a classification of what type of parallelism is offered by which system. *DCS* are suitable for *job* and *task* level parallelism. This is due to the low degree of coupling in the hardware. As the degree of coupling increases, the granularity of the application decreases. This is because of the obvious need to take the advantage of the hardware architecture. Computer networks are true distributed processing systems, whereas traditional multiprocessors are true parallel processing systems. Another point of difference is that in message passing systems, the communication overhead is relatively larger than in shared memory based parallel systems, which favours coarse grain parallelism.

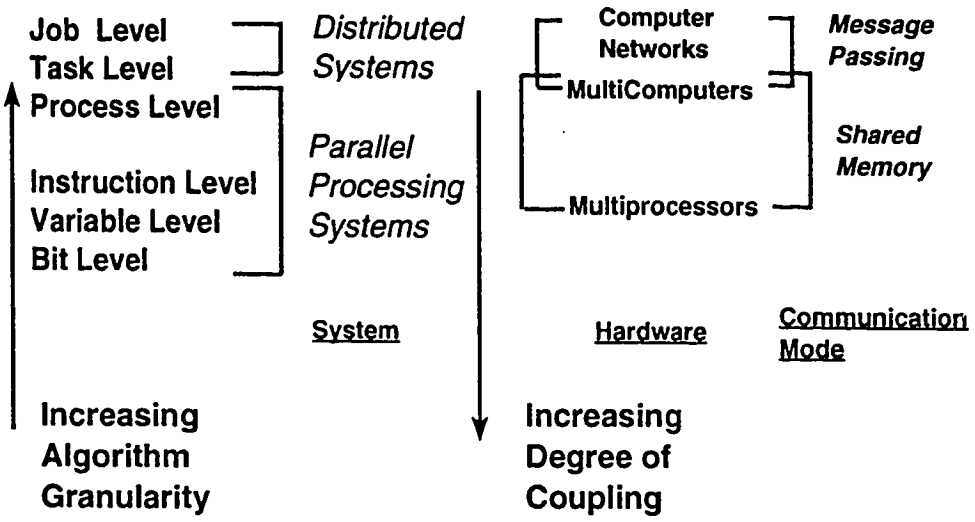


Figure 3.1: Relation between *application* granularity and parallel processing systems

Virtual Machine (VM): This is defined as a collection of hosts of the network on which the application in question is to run. The hosts in the dotted contour of Figure 3.4 represents an instance of a *VM* in a network. It is that part of the network which *hosts* the application.

Concurrency Control: This is defined as the scheme of selecting modules for execution. *LBFW* assumes that this is specified by the user. Thus support for a variety of such schemes is provided like data driven, control driven, synchronized or demand driven.

Application Communication Geometry: This is defined on the interconnection pattern between the computational modules. *LBFW* supports applications having both *regular* as well as *irregular* patterns of communication.

Routing Mechanism: This is defined as the protocols used by each node of a

network (Γ_i) to communicate with another node of a network (Γ_j). In the case of a single network it may not make much difference, but in the case of multiple interconnected networks, the routing mechanisms do become an important factor.

3.2 Requirements Specification of LBFW

3.2.1 System Model

This section gives a broad overview of the various components of *LBFW*. There are two distinct points of views in this regard: User's point of view and designer's point of view.

3.2.1.1 User's point of view

In order to use any system or environment for application programming, a programmer needs to answer the following questions:

- How does one control the application execution?
- How can the application be coded to use the functionality provided by the environment in question?

LBFW uses the following approach to solve these problems. For every application to be run, a special task called *user's agent* has to be developed by the application developer. As shown in Figure 3.2, its job is to allow the user register his application with *LBFW* and request the execution of the tasks in whatever order desired. Also it indicates *logical* end of the application to *LBFW*.

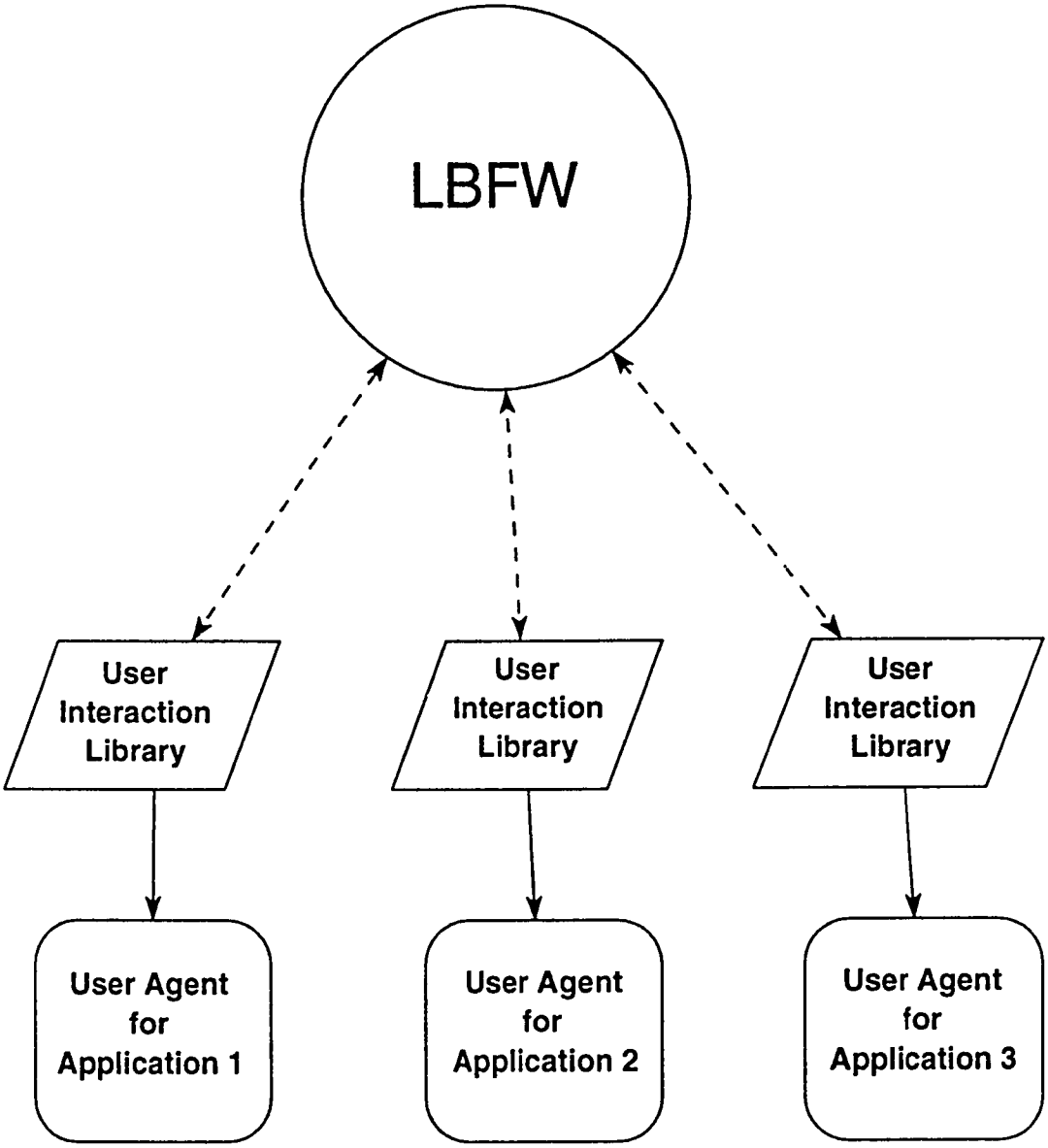


Figure 3.2: A user perspective of *LBFW*

The user agent communicates with *LBFW* using an abstraction of procedure calls. In order to hide the complexity of *LBFW*, the user is provided with an interface in the form of a *user interaction library* (Figure 3.2).

Therefore, the *user's view* of *LBFW* is of a typical operating environment. In order to use the facilities provided by it, the user has to use some library of functions provided with appropriate syntax and semantics.

3.2.1.2 System Designers's point of view

The designer's view is aimed at establishing a broad picture of the system components and their relationship (Figure 3.3). Details of individual components are presented in Section 3.4. Four subsystems have been identified. They are:

Application Execution Control Subsystem(AECS): This is the heart of *LBFW*. Most of the user commands to *LBFW* as well as the core of internal decision making is performed by it. *AECS* is expected to control the *dispatcher subsystem* by commanding it to execute a given task on a particular host. It interacts with the *load monitoring subsystem* to keep an uptodate information about the current load on the network and the individual hosts so that this information could be used by the *load balancing subsystem.*

Dispatcher Subsystem(DS): This subsystem works directly under the control of the *AECS*. It has the responsibility of executing a given task on a specified host. This may lead to two forms

1. Task execution request is received for the first time. In this case the task has to be normally spawned on the said host.

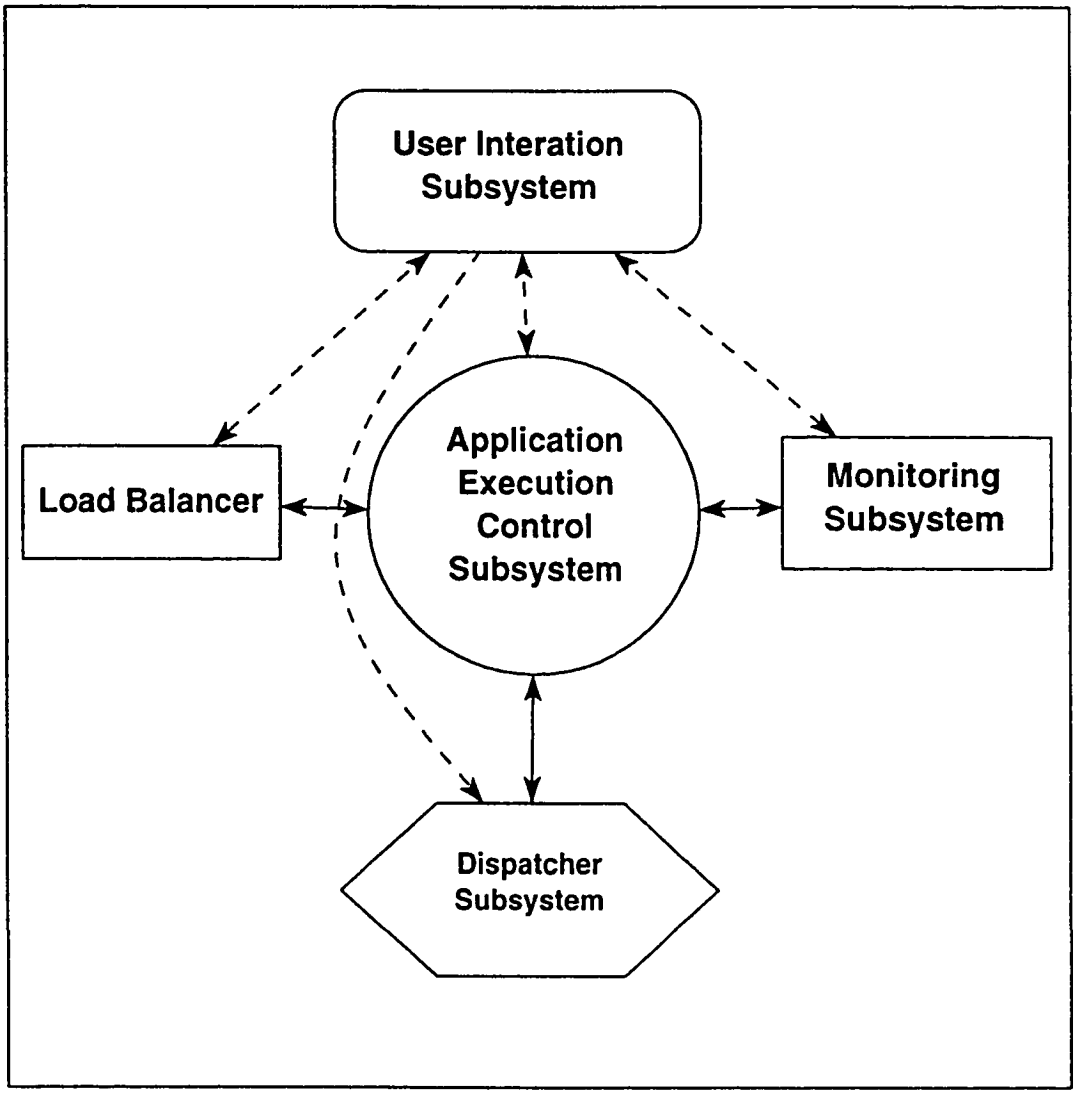


Figure 3.3: A global perspective of *LBFW*

2. Task dispatching request comes during its execution. In this situation, the dispatcher subsystem has to *migrate* the task to another machine. In this scenario, the computation already gathered by the task will have to be preserved.

Load Balancing Subsystem(LBS): This subsystem contains the actual load balancer for *LBFW*. Depending on the corresponding objective functions, this subsystem calculates the best possible mapping of tasks. Thus the load balancing provided by *LBFW* becomes *dynamic*. In addition, this can be used by the *AECS* to maintain a watch on the current execution scenario of the application and notify any need of process migration.

Load Monitoring Subsystem(LMS): This subsystem does the hard work of continuously monitoring the network and host loads and periodically keeping the *AECS* aware of the system load.

User Interaction Subsystem(UIS): This subsystem provides an interface to *LBFW*. This allows the user to concentrate on the problem. The user requests or commands are accepted in the form of procedure calls and are translated and passed in suitable format to the other subsystems and vice-versa.

3.2.2 Working Environment

3.2.2.1 System Assumptions

The following are the assumptions that *LBFW* is based on:

- a Unix based working environment,

- the parallel application is coded as a collection of executables, one for each task,
- availability of the executable code for the machine architectures present in the virtual machine,
- message based inter processor communication using standard TCP/IP based underlying communication services, and
- relatively high granularity parallel applications.

3.2.2.2 Application Characteristics

LBFW, with the current design and in the view of current technology, is more suitable for applications having a relatively high granularity. With the availability of very high speed networking, especially with the advent of Fiber Optics and ATM, the current bottleneck of transmission, routing and switching delays are expected to reduce dramatically. This, coupled with the ever increasing power of workstation allows *LBFW* support for applications with much more variation in their granularities.

Another related field that is growing very fast and may have pronounced impact on distributed programming in general and *LBFW* in particular is distributed operating systems. Because of the importance of networking in today's computing, an increasing amount of work is being done on it. One of the most important drawbacks of network of workstations is the lack of active process migration, which may be embedded into operating systems. This would again help in increasing the utility of *LBFW*.

3.2.2.3 Functional Requirements

One way of looking at the functional requirements of *LBFW* is a *stimulus/response* mode of operation. Given a particular stimulus, *LBFW* produces its corresponding response. The stimuli can be presented under two headings:

- *periodic Stimuli:* occur at predictable intervals of time and
- *aperiodic Stimuli:* occur irregularly at any point in time because of the current state of *LBFW*, the current application execution or user requests.

Such stimuli arise from the application needs. For example:

- Register or start an application : The applications must therefore be registered with *LBFW*.
- Execute tasks of an application: Once an application is registered, the user needs support to request the execution of the tasks of the application in any order.
- Send and receive data: To provide inter-task communication.
- Terminate an application : Once the *logical* ending of an application is reached, there is no need to maintain all the overhead information about the application in the memory.

3.2.2.4 Main Constraints

There are a few conditions under which *LBFW* operates.

- Since *LBFW* is intended for a forest of workstations based computing environment, heterogeneity of hardware and software is inevitable. Thus *portability* is of vital importance.

- Active process migration is a very difficult problem. One of the reasons for this is environment heterogeneity. Another reason could be the current status of the networking technology. Therefore good performance of load balancing has to start with a good initial placement.

- The response time of the load balancer is of crucial importance. The benefits of a good assignment for a particular system state might be lost if the load balancer takes a long time compared to the time it takes for the system state changes.

- Dynamic changes in system load and especially network load is also an important issue. The methods for having a completely uptodate information about the system load is an exorbitantly expensive and practically almost impossible. Thus, the assumption that the system load is reasonably constant for a time period is inevitable. That is changes in load in this interval, will not be reflected in *LBFW* decisions.

3.3 Architectural Design

Based on the discussion in section 3.2.1.2, a resultant architectural design of *LBFW*, Figure 3.4, shows a virtual machine with three user applications. It shows three interconnected *LAN* based networks, one Ethernet based LAN, a Token ring and

an FDDI based LAN. The hosts in the dashed contour form the *VM* or *DCS* of interest.

LBFW consists of the following active components integrated to realize the users and the designers view.

Application Controller: It integrates the functions of the *AECS*, *LBS* and *UIS* (Figure 3.3). It uses centralized decision making.

Application Agents: They do the job of the *DS* in Figure 3.3.

Load Controller: It plays the role of a part of *MS* in Figure 3.3. It gathers system load information from the various *load agents* and periodically updates the *application controller* about it. This design decision was taken in order to make the *application controller* as fast as possible. Now the application controller takes the system load from one source rather than N sources where N is the number of hosts in the *VM*.

Load Agents: They perform the actual gathering of system load and report it to the *application controller*.

At any time during the execution of *LBFW*, only one copy of the application controller exists. The host running the application controller is called *control host*. On each node of the *VM*, an application agent and a load agent exist. A load controller resides on the *control host*, with the application controller.

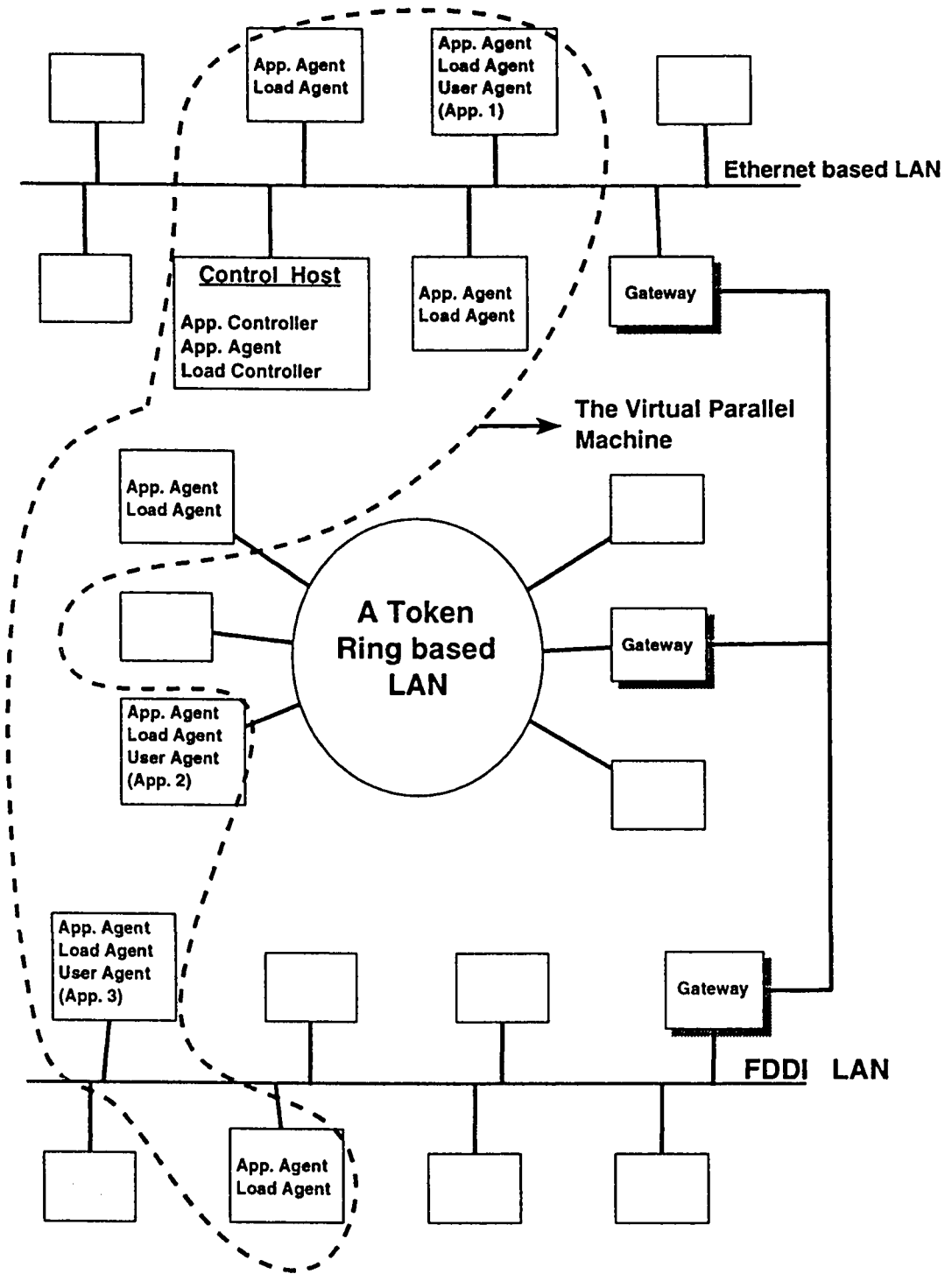


Figure 3.4: An Illustration of Architectural Design of LBFW

3.4 Detailed Design

For each *LBFW* component, its functionality, the design of data structures used by it and its interface with the rest of the system are discussed.

3.4.1 Application Controller Design

This is the heart of *LBFW*. In line with the design of most of the network based distributed applications, the design of *LBFW* is of *Client-Server* type. Application controller is the *compute-server* for *LBFW*, any request for computational resource must go through it.

Figure 3.5 shows the internal structure of the application controller. It is realised by a number of sub-modules.

3.4.1.1 Startup Module

Startup module is responsible for the proper initiation of *LBFW*. It takes charge during *LBFW* boot-up and is responsible for getting the *LBFW* components up and running and ready to honor any request for application execution, functionality of which is specified by Algorithm 3.1

Algorithm 3.1 StartupModule

```
Find the number of hosts in the current Virtual machine(VM)
Foreach host in VM do
    Setup an application agent
    If (CurrrentHost == controlHost) Then
        Setup the load controller
    Else
        Setup a load agent
```

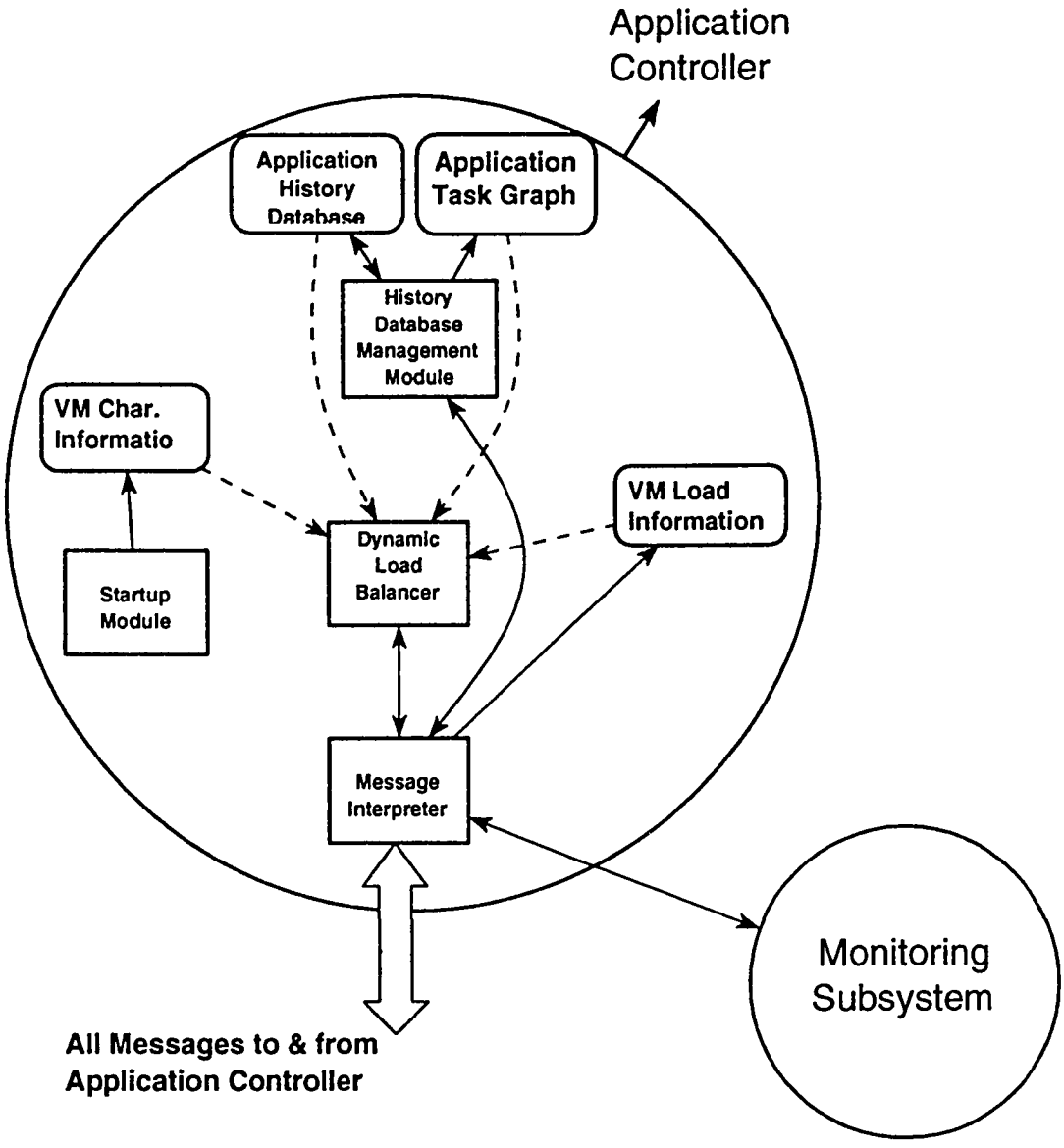


Figure 3.5: Block diagram of the Application Controller

Endif
EndFor
Force an initial load exchange so as to come to a stable state
End

Interface

Startup module does not actively interact with any other subsystem. Its job is to set the system running for the first time after which it goes into oblivion.

Data Structures

Since this module is executed once during the boot-up of *LBFW*, it sets up some global data structures. An important data structure is the *VM Characteristic Information* which provides information about the machines present in the *VM*. This is a critical information about the *capabilities* of the hosts for that session of the *LBFW*, which is frequently used by the load balancing module.

The *VM* characteristic information is read from a specification file, assumed to be defined by the user. A sample specification file is shown below.

```
#Name      Processor No Clock RAM  Swap  MIPS
#-------------------------------------------------------------------
#########  NeXT  Machines  ##########
makkah      m68040   1  33    64    32    1
madinah     m68040   1  33    64    32    1
jubail      m68040   1  33    16    32    1
jeddah      m68040   1  33    16    32    1
riyadh      m68040   1  33    16    32    1
dammam      m68040   1  33    16    32    1
khobar      m68040   1  33    16    32    1
abha        m68040   1  33    16    32    1
#########  SUN4  Machines  ##########
bayadh      sun4     1  66    32    32    1.5
sijan       sun4     1  66    16    32    1.5
canad       sun4     1  66    16    32    1.5
```

```
farsi       sun4     1  66     16     32     1.5
shaoor      sun4     1  66     16     32     1.5
```

The format of a specification file is self explanatory. The point to be noted here is that the algorithm which calculates the capability of the machines is used once during the bootup process, as the *VM* is assumed to be fixed, during the execution of *LBFW*. This information is compiled in the form of a global array of records called the *VM Characteristics Database*.

3.4.1.2 Message Interpreter Module (MIM)

MIM is the main workhorse of the application controller. It contains the main work loop of *LBFW* and does the job of sending and receiving messages to the outside world. *MIM* is referred to as *interpreter* because it understands the messages and makes sure that appropriate actions are taken for each message arrived. In a distributed application like *LBFW*, the importance of such an entity is obvious. But this might also become a serious liability, because too many messages to the application controller are bound to slow it down which will have direct impact on the performance of the parallel applications running on it, defeating the whole purpose. Thus, one of the major design issues is reducing the number of messages handled. Algorithm 3.2 discusses the functionality of this module.

Algorithm 3.2 MessageInterpreter

```
Loop
    Case (Request) Of
        STARTAPP:       SetupApplicationEnvironment;
        ENDAPP:         SaveApplicationEnvironment;
                        ReleaseMemoryHeld;
        EXECUTETASK:    LoadBalancedSpawn;
        LOADREQ:        SendCurrentLoadInfoToCaller;
        FINISHED:       UpdateApplicationEnvironment;
        CURRENTLOAD:    UpdateCurrentLoadInfo;
        RESULTS:        PassTheResultToUserAgent;
    EndCase
EndLoop
End
```

Interface

MIM has to interact with almost all the other modules because of its strategic location. It interacts with the *history management module* for setting up the application environment and maintaining it. It interacts with the *load balancer* for executing a given task on an appropriate machine. It interacts with the *load controller* subsystem to obtain the current load request from it.

Data Structures

This module does not posses any major data structure for itself. It merely access information in response to user commands.

3.4.1.3 History Management Module

In order to make an effective load balancing decision, a load balancer needs to know the resource requirements of the tasks in question. Most of the load balancing

heuristics in literature assume *somehow* the availability of the execution times and other metrics about the tasks. This is a an unpragmatic approach.

In this study, it is assumed that the only information known to *LBFW* the first time any application runs, is its directed task graph and system data. On each run, statistics about the execution time of tasks and the communication volume between them is measured and maintained as the history of application. This information is used by the load balancer during all later runs, for making a load balancing decision using an appropriate prediction function.

Two important problems arise when history mechanism is used.

- What if two users use a same application name for their different applications?
- What if two users use different names for the same application?

In the former case, a simple solution could have been to outrightly restrict such cases by giving appropriate messages to the user. But the latter still remains unresolved. In order to solve this problem, a *History Mapping Mechanism* is introduced. The basic function of this module is to map the user supplied application names to unique names according to which the history information is stored (Figure 3.6).

In order to map the application names to a unique one, application characteristics are used to differentiate between different applications. The question of which characteristics to choose is an implementation decision and is dealt with in detail in Chapter 4. Some of the candidates are the sums of sizes of all the executable of an application, extracting part of header information from each executable and so on.

The basic idea is as follows: Let f be a function defined as $f : A \times B \longrightarrow C$ where A is the set of all possible application names. B is the distinguishing characteristics of the application and C is the set of *unique* application identifiers.

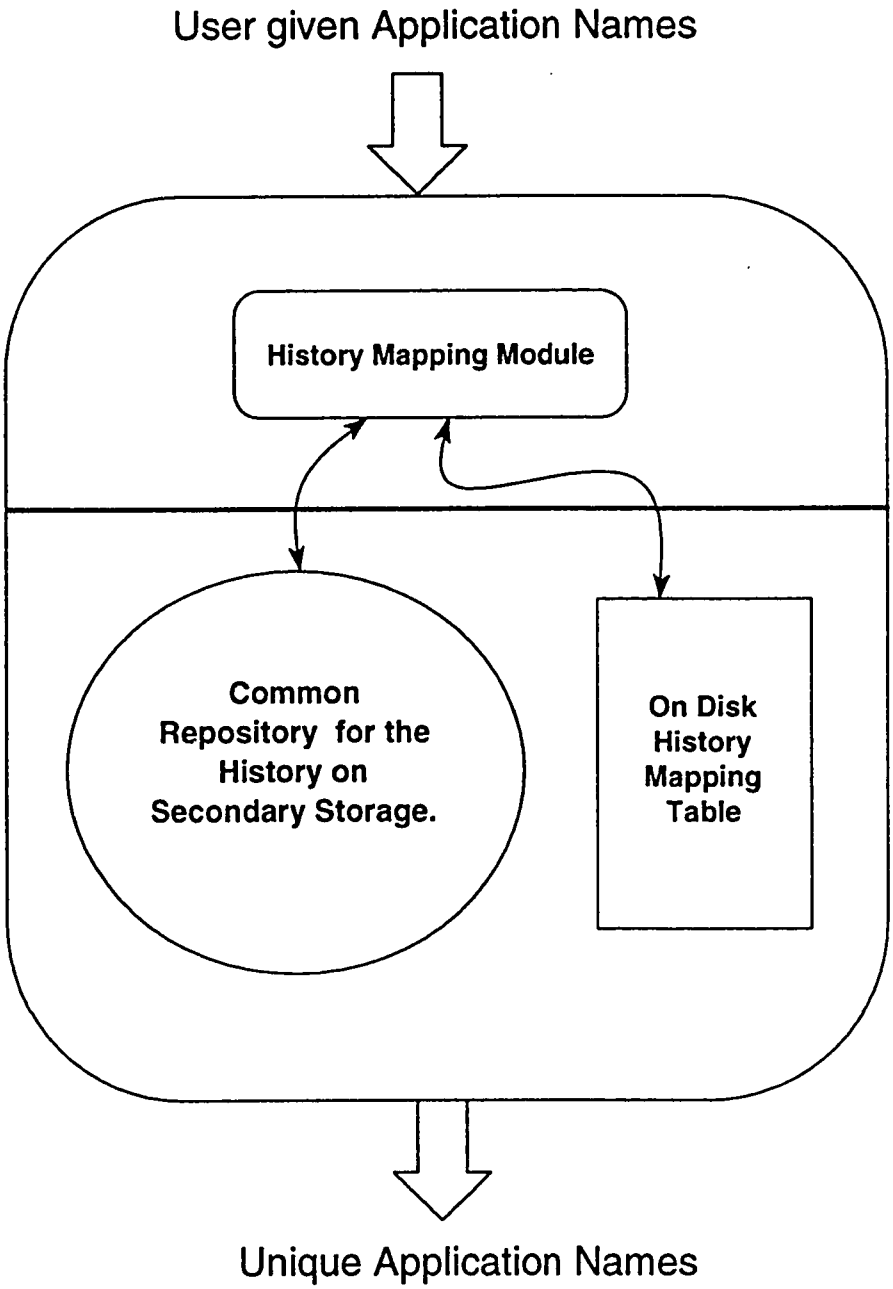


Figure 3.6: Block diagram of the History Mapping Mechanism

Figure 3.6 elucidates the above idea in the context of *LBFW*. The secondary storage of history information contains, for each registered application, the task execution times and the communication volumes. The mapping table contains a set of records of the form

⟨*Application characteristics, Unique application identifier*⟩ .

Thus, if an application, named α by the user, application characteristics, β, could be extracted from it. The unique application identifier can then be generated as *unique identifier* = $f(\alpha, \beta)$. The mapping has to be applied each time a task refers to the application using the programmer's given name, transparent to the user.

Interface

The history information is needed by the load balancing subsystem. The application controller, should be able to load or update history information whenever a task is started or finished.

The History Data Structures

The features expected of history data structures include

- support for multiple applications running concurrently,
- support for applications of any type of task interconnection,
- easy access to relevant information and
- efficiency.

Figure 3.7 gives the data structure used by the history management module. It shows a snapshot of *In-Memory* history information maintained by the history

management module when three user applications are running. The applications are represented in the form of square boxes and are labelled as *Application1* thru *Application3*. A list of applications is maintained. Associated with each application is a list of tasks. This is represented by the round edged boxes and labelled *Task1*, *Task2* etc. *Application1* in Figure 3.7 has four tasks. Each task may communicate with one or more of the application tasks. These are represented by the parallelograms. *Application3* receives information from *Task2, Task3 and Task4*, whereas *Task2* in the same application does not receive information from any task. The data structures are versatile in handing application task graphs of any kind.

Each of the *application* boxes has the following data structure:

```
typedef struct applicationExecutionInfo {
    char applicationName[256];
    int userAppTid, taskCount, firstRun;
    TASKINFO *listOfTasks;
    struct applicationExecutionInfo *next;
    struct applicationExecutionInfo *prev;
}APPINFO;
```

The data structure for each task is as follows

```
typedef struct taskExecutionInfo {
    char taskName[256];
    int nRuns;
    long avExecTime;
    double avLoad;
    double load;
    COMMINFO *communicationsList;
    struct taskExecutionInfo *next;
    struct taskExecutionInfo *prev;
```

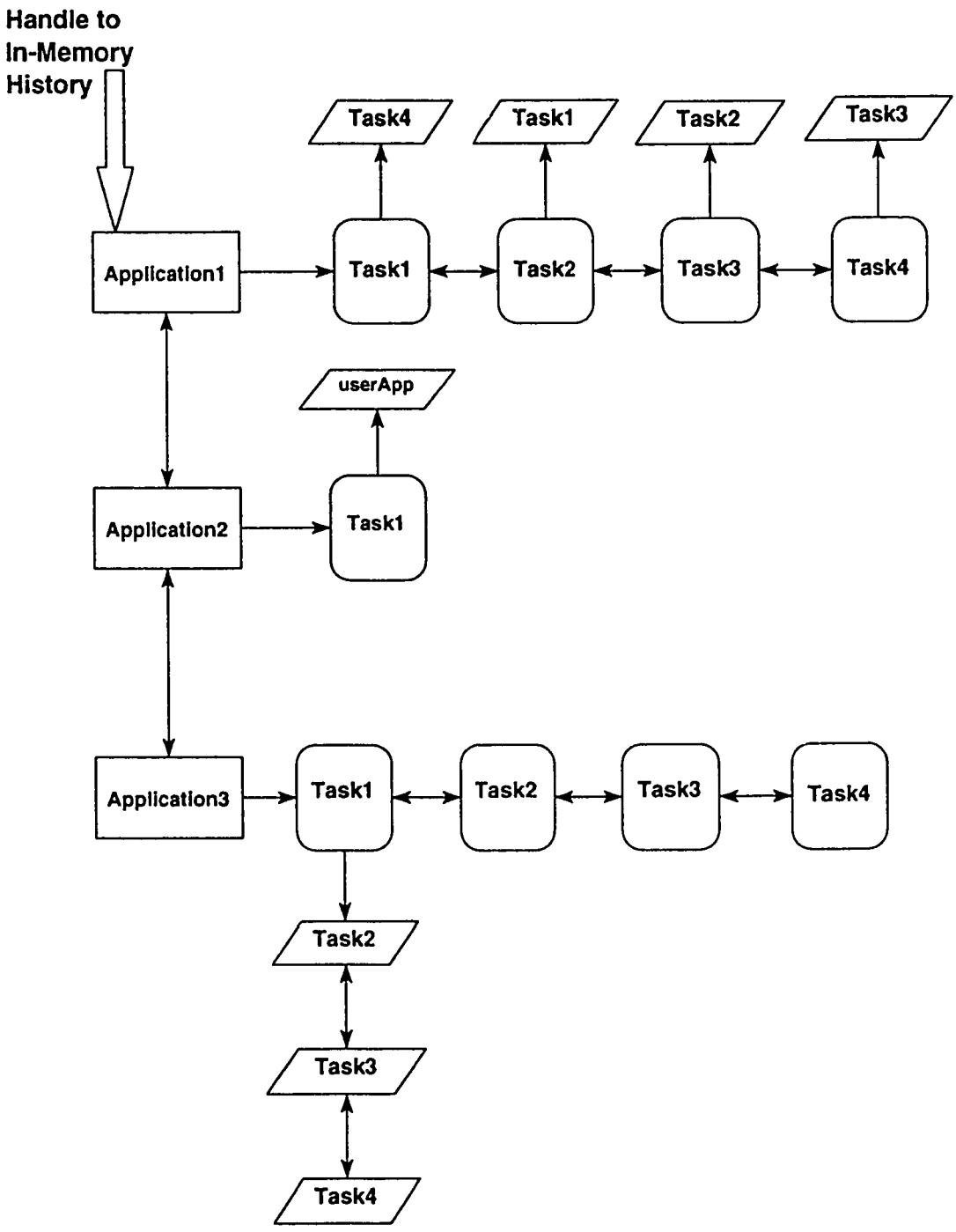


Figure 3.7: Data Structure diagram of the History Mapping Mechanism

```
}TASKINFO;
```

The data structure which maintains the amount of data received by each task is as follows

```
typedef struct communicationInfo {
    char taskName[256];
    long maxBytes;
    long minBytes;
    struct communicationInfo *next;
    struct communicationInfo *prev;
}COMMINFO;
```

3.4.1.4 Load Balancer Module

Objectives of Load Balancing

The most pressing issue in the design of a load balancing methodology is the necessity to define a proper objective function for it. Much was thought about what this objective function could be like. There are three main alternatives:

- *Balanced Resource Utilization:* Here, the basic idea is to minimize the idle times of workstations. As long as the workstations are busy doing something, it is fine. In fact, they may be doing things for which they are not very well suited. This is the major drawback of this objective function. From the point of view of a parallel applications, this concern, of course important, has a secondary status to the application characteristics on hand. The *Greedy* based algorithms fall under this category.

- *Minimization of Response Time:* This objective function aims at reducing the response time of the parallel application. The problem with this is in order to minimize response time, some of the better hosts might be more heavily utilized than the others. For example a host could be used for executing more tasks because of higher communication overheads otherwise. This could lead to a load imbalance among the workstations in the case where there are large number of tasks with high granularity, in an application.

- *Minimization of Completion Time or the Maximization of Throughput:* The aim is to minimize the completion time of every task when assignment made such that the desirable objective of minimization of response time of the application is met.

3.4.1.5 A Load Balancing Heuristic - lbHeuristic

The load balancing heuristic used considers minimization of completion time (or maximization of throughput) for every task assignment, as its objective function. It is based on having a good initial placement of the task. This is because the problem of active process migration is a difficult one and is out of the scope of this study. *lbHeuristic* takes the following items as inputs:

- the tasks of the parallel application, their completion and communication times

- information about the workstation types and their suitability for different types of tasks. For example some of the hosts could be parallel processors themselves.

- A measure of the relative distance between the workstations and a measure of the current load on the virtual machine.

Total Cost:

Let an application be represented as $A\langle T, E, C, N\rangle$ where

$T = \{t_1, t_2, t_3, t_4, \cdots, t_n\}$ represents the tasks of the application

$E = \{x_1, x_2, x_3, x_4, \cdots, x_n\}$ represents their execution times and

$C = \{c_{ij}|$ communication cost between task t_i and $t_j\}$ If the tasks do not communicate then c_{ij} is $0(zero)$. Using this information, the total cost of the application becomes:

$$Total\ Cost\ =\ \sum_{i=1}^{n} x_i + (\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}) \tag{3.2}$$

Minimization of the above cost have been explored by many researchers. Jack Worlton [26] has studied the limits of parallel computations. He assumed that a parallel program typically consists of repeated instances of synchronization tasks followed by a number of actual computational tasks distributed over the VM. He assumed that all the processors had the same capacity. This is not true in our case. Assuming that the difference is not drastic, because of various overheads, the total cost on a parallel system is longer than it would be if they were executed on a single processor, assuming communication is nil.

Parallel Cost:

Let

t_s = synchronization time

t = task execution time

t_0 = task overhead caused by parallel execution

N = number of tasks

P = number of hosts in the VM

In a parallel environment, each task requires $(t + t_0)$ time units rather than just t. For N tasks executed on P hosts, the number of parallel steps is the ceiling ratio $\lceil N/P \rceil$. The parallel cost may be approximated as follows

$$T_{(N,P)} = t_s + \lceil N/P \rceil (t + t_0) \tag{3.3}$$

Even though this is the case with parallel cost, the response time or the completion time can be reduced due to notion of parallelism, i.e. multiple tasks of the same application execute concurrently and independently.

Let s_j be the speed of host j (normalized with reference to a typical processor) and l_j be its load at that instant. Further let l_j be a value between 0 and 1. If $l_j = 0$ then the processor is fully free and if $l_j = 1$ then the processor is fully busy. Then the expected completion cost of a task i on host j, having an average execution time x_i is

$$Completion\,Time(task_i) = x_i s_j l_j + \forall_k \sum c_{ik} D_{\alpha(i)\alpha(k)} \tag{3.4}$$

where c_{ik} is the communication cost acrued when any task k communicates with task i,

$\alpha(i)$ is the host where task i is running,

$\alpha(k)$ is the host where task k is running and

D_{pq} is the *distance* between host p and host q. This is defined as a function which returns the time it takes to send a unit amount of data between p and q.

3.4.1.6 Load Balancer Module - Interface

One if the important design features of *LBFW* is that it is independent of a specific load balancing policy. The interface between this module and the others are defined as in Figure 3.8.

The inputs to the load balancer are the task name, application name, arguments and the number of instances. In addition, the application execution times and the communication times are taken from the history database and the *VM* load information is fed from the monitoring subsystem. The output of any load balancing heuristic, in general, is a sequence of triples ⟨ *Task*, *Host*, *Number of Instances* ⟩. This in the *LBFW* context is termed as the *Spawn-List*. Therefore from the point of view of the application controller, any load balancing policy generating the *Spawn-List* in this format is fine.

3.4.1.7 Load Balancer Module - Data Structures

As discussed in the previous section, the most important data structure that this module generates is the *Spawn-List*. This information is utilized by the application controller to allocate tasks to hosts. Once the decision as to which task should execute where is already made, by the load balancer, the semantics associated with the *Spawn-List* become very simple. Each triple denotes an assignment of the *Task*

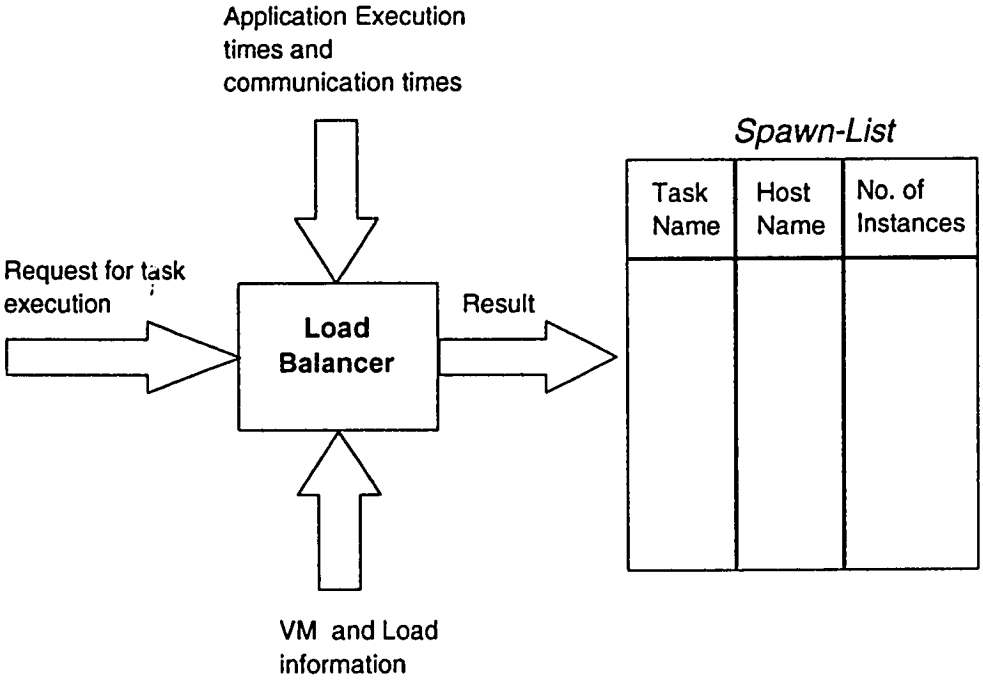


Figure 3.8: Block diagram of the Load Balancer Interface

to a *Host* (Figure 3.8). The semantics are captured by the following data structure.

```
struct spawnInfo {
        int hostId;
        int nInstances;
        char *taskName;
        char **argVector;
};
```

Apart from this the load balancer module also needs to access data from the history management module, the virtual machine characteristics database, the virtual machine load information and the application task graph. The information regarding application task graph and its history is obtained using the global handle to *In-Memory* history structure of Figure 3.7. Similarly the *VM* characteristics database is a global array of records (see Section 3.4.1.1). The *VM* load information is obtained by the global array maintained by the monitoring subsystem (Section 3.4.3.1).

3.4.2 Application Agent Design

Application agents are proxies of the application controller on each host of the *VM*. They receive commands from the application controller and do the job of actual execution and track the execution of user applications. It contains three major modules, the message interpreter module, task execution monitoring module and task dispatcher module. Figure 3.9 shows the internal structure of the application agent.

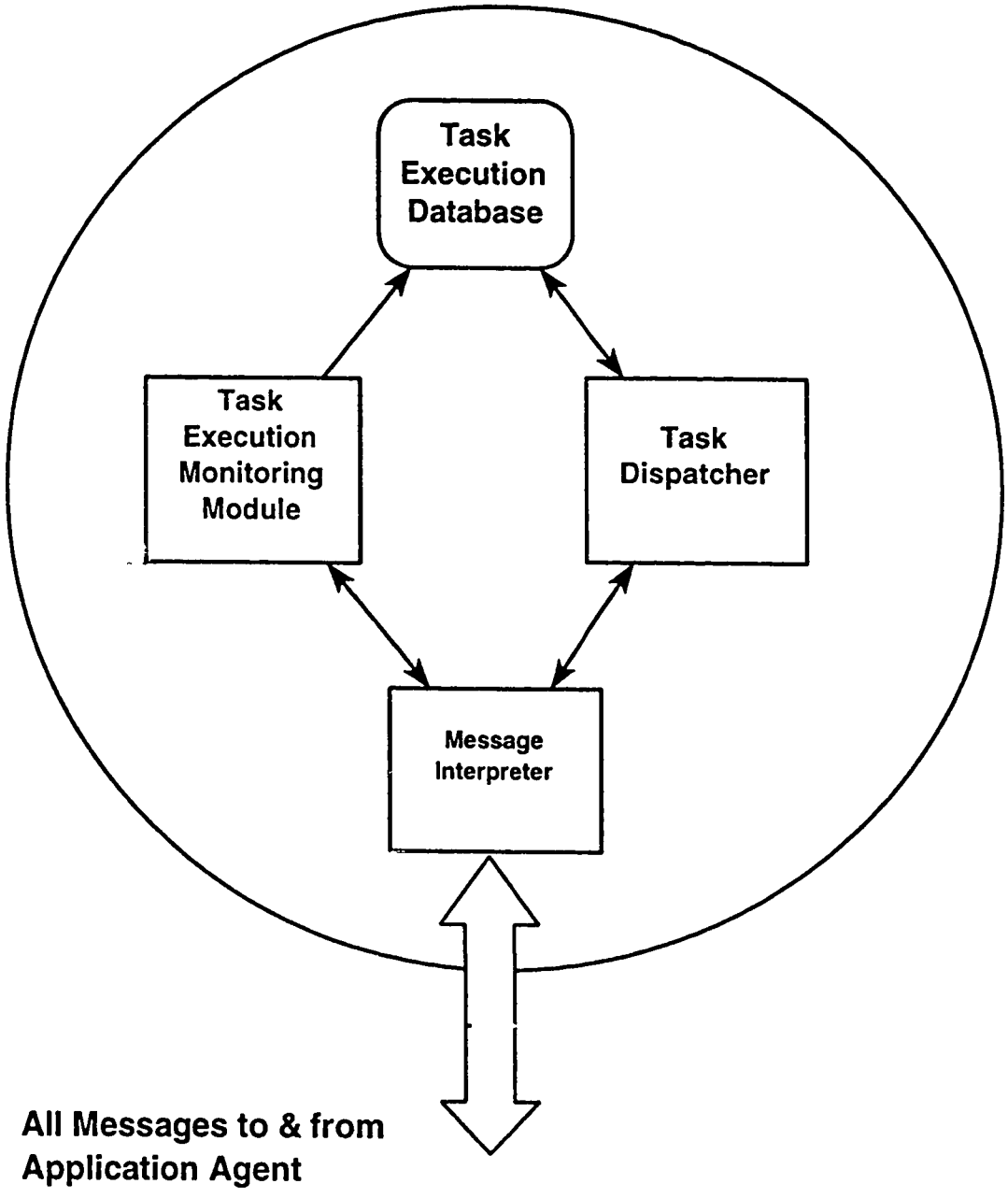


Figure 3.9: Application Agent

3.4.2.1 Message Interpreter Module

This module has a functionality similar to the message interpreter module of the application controller. The only difference being that the former has a global view of things the latter has a local view. The functionality can be summarized by the Algorithm 3.3.

Algorithm 3.3 MessageInterpreter

```
Loop
   Case (Request) Of
      MIGRATE:            MigrateTask;
      EXECUTETASK:        ExecuteTask;
      TASKTERMINATED:     UpdateProcessStructure;
      COMMUPDATE:         UpdateCommunicationVolume;
      UPDATETIME:         SendCommInfoToAppController;
   EndCase
EndLoop
End
```

Interface

This module has to interact with the other three modules of the application agent. It interacts with the *task execution monitoring module* for setting up of the task execution data structures and maintaining it. It interacts with the *task dispatcher* for executing a given task and with the *application controller* to obtain the next command to execute.

3.4.2.2 Task Execution Monitoring Module

There could be many applications running on the virtual machine. Each application usually has many tasks. In order to have better control over the execution, it is

necessary that the information be maintained wherever the task is executed. This module maintains such data and forwards it to the application controller once a task terminates.

The module also maintains the mechanism for migrating active processes when there is a need. The evaluation of *when there is a need* is done by the application controller. A *Checkpoint* and *Restore* based method for active process migration in the context of workstation based networks is described in [27].

Another important functionality of this module is its recording the exact amount of data (in bytes) received by a task executing under its control from any other task (in a weird case of any other application). This is done transparent to the user, by the send and receive routines acting as trojan horses for obtaining this information.

This module is also given the responsibility of finding the execution time of the tasks. When a task terminates, it is the function of this module to to send all the gathered statistics to the application controller.

Interface

This module interacts with the message interpreter module and provides access to its data structures.

Data Structures

The data structure provided and maintained by this module is termed as *task execution database*. Since it is recognized that many applications may be executing simultaneously on the virtual machine, there is no guarantee that the tasks executed by the application agent all belong to the same application. Another factor is that the number of tasks from which a task receives information in the form of messages is variable. The task execution database is designed to provide this information.

Each task is recognized by a unique task id, which is shown in Figure 3.10.

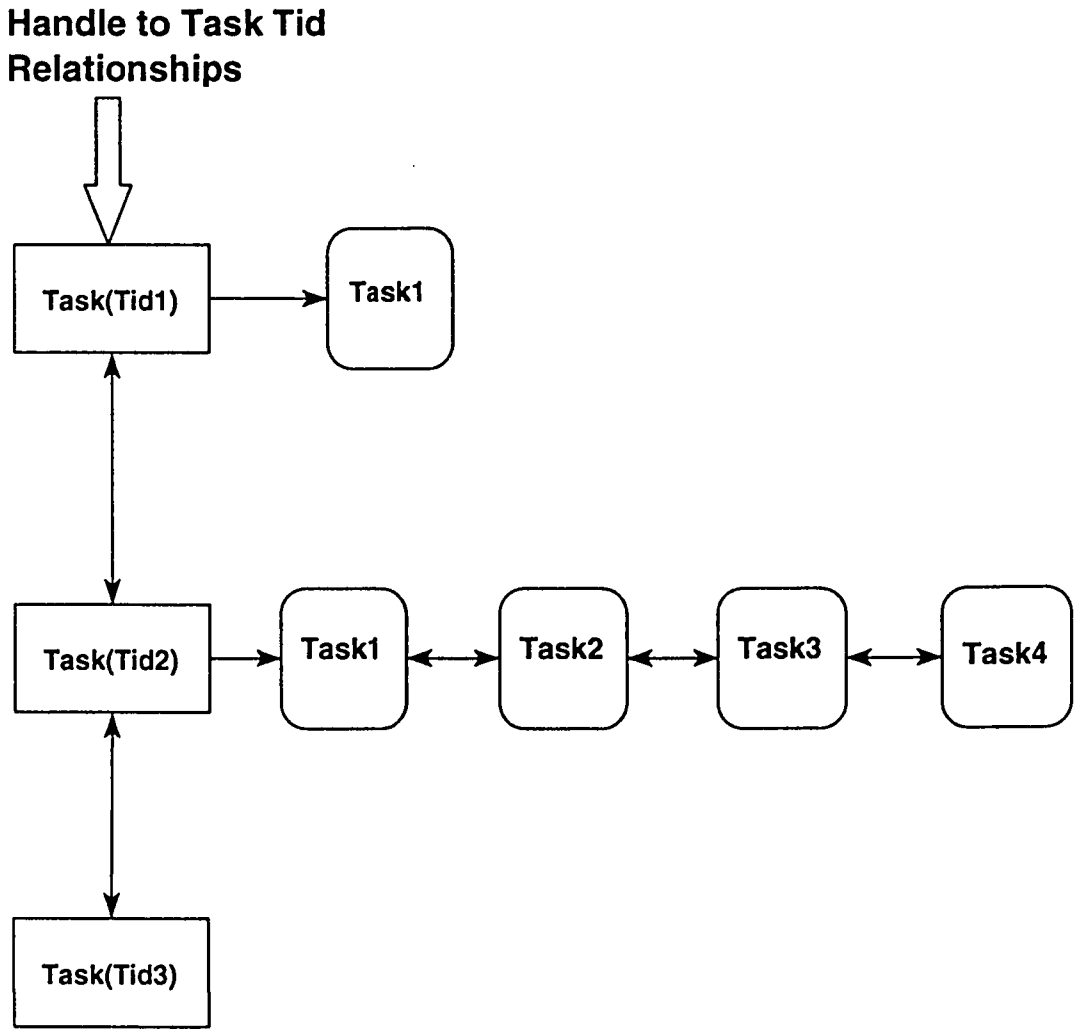


Figure 3.10: Task Execution Database Structure.

Each square box denotes a task executing under the control of the task execution control module. It is uniquely identified by a unique task identifier and is further qualified by the unique application name generated by the application controller. The round square boxes identify the tasks from which the task has received any

information and the quantity.

3.4.2.3 Task Dispatcher Module

This module executes a given task on the current machine. It unpacks the arguments and passes them to the task as dictated by the user. This interacts with the message interpreter directly and with the task execution monitoring indirectly through the task execution database.

Data Structures

This module creates a task node of the task execution database when it dispatches a task. It means that one square box in Figure 3.10 is created per task. The information maintained therein is summarized as follows:

```
typedef struct taskTidRel {
        int tid;
        int deletedFlag;
        char taskName[256];
        char appName[256];
        struct timeval *startTime, *endTime;
        int commCount;
        COMMLIST *commList;
        struct taskTidRel *next;
        struct taskTidRel *prev;
}TASKTIDMAP;
```

3.4.3 Monitoring Subsystem Design

This component of *LBFW* monitors load information on the current virtual machine. The monitoring subsystem is composed of a central load controller and distributed load agents one per host.

3.4.3.1 Load Controller

There is one load controller component in *LBFW*, located on the *control host* in order to provide only one input point for load information to the application controller. It accumulates the load information from the virtual machine in a suitable format and present it to the application controller periodically or upon request. The function of load controller is stated in Algorithm 3.4.

Algorithm 3.4 LoadController

```
    Loop
        Case (Request) Of
            LOADSTATUS:     RecieveLoadInfoFromLoadAgents;
            GETLOAD:        GiveNormalizedLoadToAppController;
            CURRENTLOAD:    GiveNormalizedLoadToAppController;
            GIVELOAD:       GetLoadFromLoadAgents;
            UPDATETIME:     GetCurrentHostLoad;
        EndCase
    EndLoop
End
```

Interface

Load controller is the representative of the monitoring subsystem for the application controller. *LBFW* is unaware of any load statistics maintained by this subsystem. The interface between load controller and *LBFW* is based on the former providing normalized load values for the *VM* to the latter at regular intervals of time or on request.

Data Structures

Load controller maintains the load values for each host in the form of an element

of a global array. This information seen by the application controller as the virtual machine load information database.

3.4.3.2 Load Agent

Load agents are the proxies of the load controller on each host of the virtual machine except the control host. The job of a load agent is to, periodically, access load information from the virtual machine and let the load controller know about it. The functionality of the load agents is expressed in Algorithm 3.5.

Algorithm 3.5 LoadAgent

```
Loop
    Case (Request) Of
        GIVELOAD:       SendCurLoadToAppController;
        LOADSTATUS:     SendCurLoadToAppController;
        REFRESHTIME:    SendCurLoadToAppController;
        UPDATETIME:     GetCurrentHostLoad;
    EndCase
EndLoop
End
```

3.4.3.3 Load Agent - Interface

Load controller interacts with the actual virtual machine on one hand and the load controller on the other hand. This part is completely unknown to the application controller. Hence. the flexibility of having different ways of load gathering and presenting to the load controller.

3.4.4 Component Interaction Protocols in LBFW

This subsection deals in detail with the protocols of interaction among the various entities of *LBFW*. Any distributed algorithm is based on communication of messages between its tasks. Since the tasks are distributed in nature, in order to solve the problem on hand, it is necessary that a receiver expects what sender has sent and takes necessary action based on the message. Thus, a protocol is defined as a set of rules followed by communicating tasks to complete their interaction. The details of all messages and their semantics are presented in Appendix A.

3.4.4.1 Boot-Strap Protocol

The bootstrap module is responsible for starting up *LBFW*. It is a critical module as it is the one which distributes the various entities of the framework across the *VM*. It is not necessary that always one particular host acts as an application controller. Any host which starts the framework is deemed to be the *control host*. The startup protocol then instantiates the *load controller* on the same machine as the application controller, so as to speed up the communication between them. It then sets up the *load agents* and the *application agents* on all the nodes of the *VM*. Since the *load agents* need to report to the *load controller*, it broadcasts the id of the *load controller* to the load agents. The reverse communication is also needed. Therefore the ids of all the *load agents* are sent to the *application controller*. For an initial load estimate load information is provided by the *load protocol*.

The messaging involved in the bootstrapping *LBFW* is shown in Figure 3.11. The message ordering is indicated by the tags in increasing order. After starting up all the components, the application controller sends the $CENTRALLOADMODULEID$

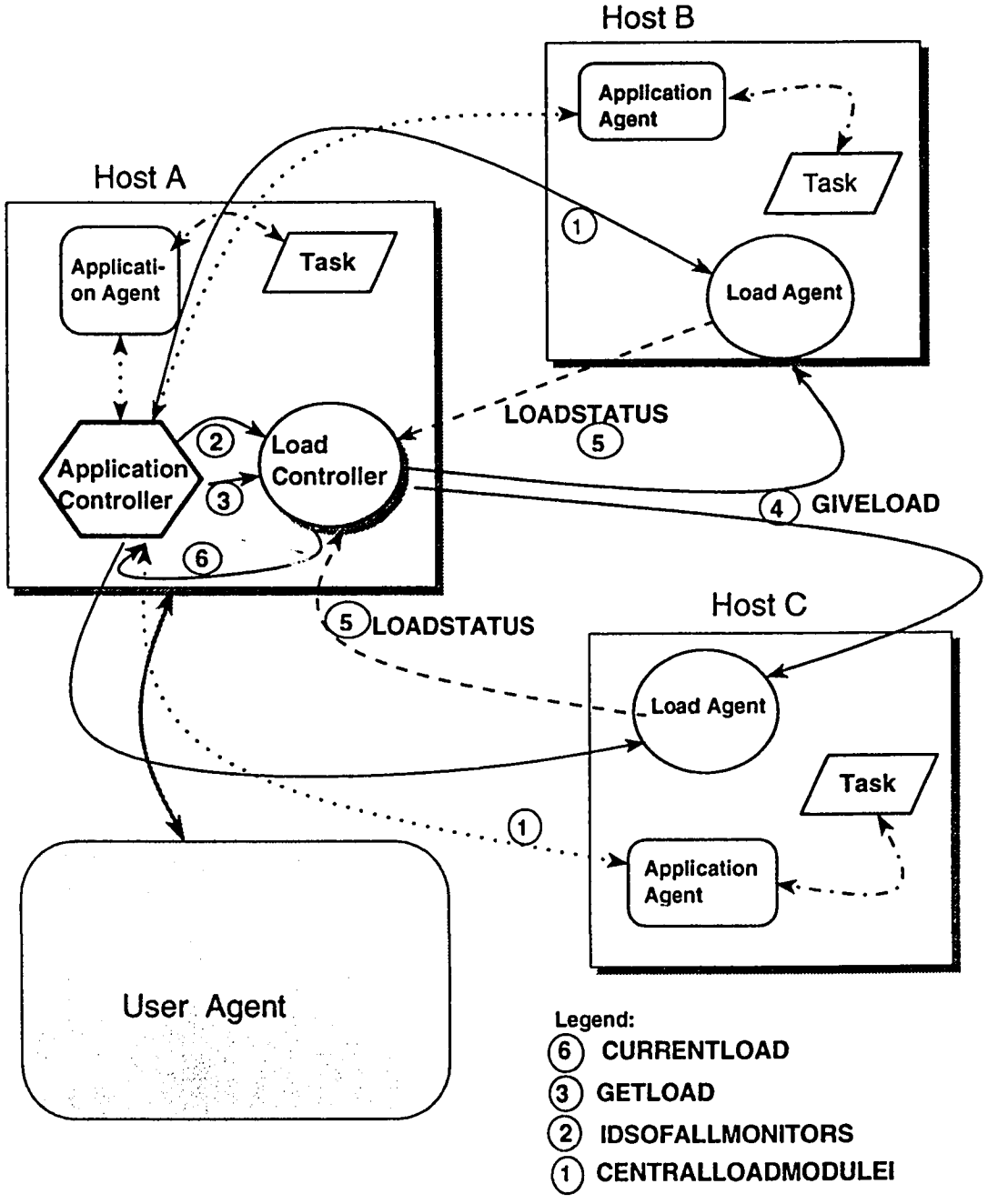


Figure 3.11: *LBFW* System Startup

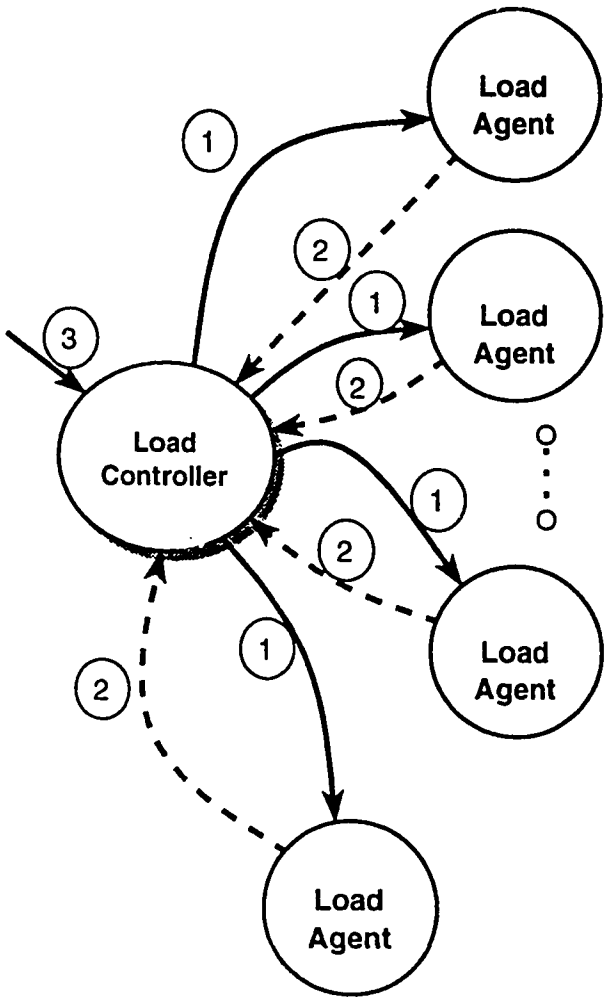
message to the load agents (message tagged as 1(one)) Figure 3.11. Then supplies the ids of all load agents to the load controller using the *IDSOFALLMONITORS* message. Then the initial load protocol is started by the *GETLOAD* message to the load controller, to which the reply is sent in the form of the *CURRENTLOAD* message.

3.4.4.2 Load Protocol

Load protocol is used by the *load controller* and the *load agents* to keep the load information uptodate. The concept of *load* is understood as an up to date view of the amount of work being done by a particular processor which varyies temporally. The current implementation measures the load on a host as the *Unix* one, five and fifteen minute load averages. This is directly proportional to the average number of processes in the ready and run queues during the last one, five and fifteen minutes. The *load agents* periodically collect this information and send it to the *load controller*. Alternatively, as in the case of Figure 3.11, the load information is given to the load controller upon request from it. Figure 3.12 shows the important messages involved in this protocol.

3.4.4.3 Application-Load Controller Protocol

Since application controller makes the load balancing decisions, it needs uptodate load level of the *VM*. It does so using this protocol. This protocol has to provide the flexibility of tuning the performance of the monitoring sub-system according to the application characteristics in addition to the default mode of operation. For example, frequent monitoring may be unnecessary if the overall performance is satisfactory



Message Description

(1) *GIVELOAD*

Source: Load Controller
Destn: All Load Agents
Semantics:
Sent when the source needs to have updated load values from the destn.

(2) *LOADSTATUS*

Source: Load Agent
Destn: Load Controller
Semantics:
Sent either when
- Source receives a GIVELOAD message or
- When the load update time is up.

Data:
Unix one, five and fifteen minutes load averages

(3) *External Messages*

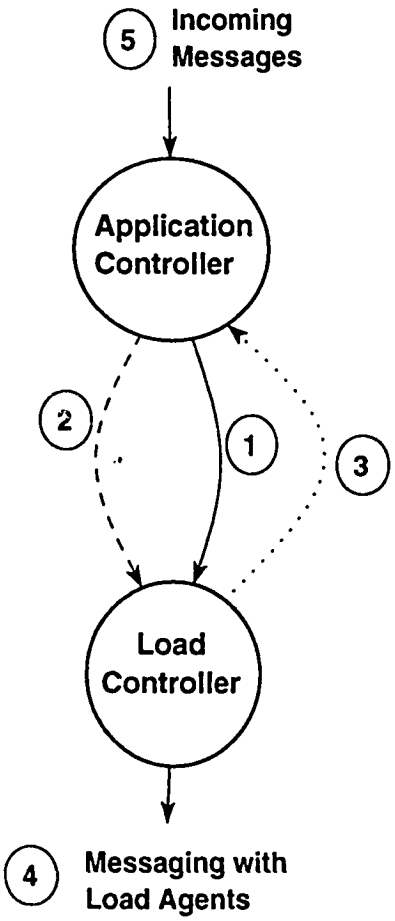
Semantics:
Handled by the Application-Load Controller protocol

Figure 3.12: Messaging in the Load Protocol

with infrequent monitoring. Figure 3.13 shows the important messages involved in this protocol.

3.4.4.4 Application Controller-Agent Protocol

This protocol coordinates the execution of the user application in a distributed manner. It involves the interactions of *application controller* and the *application agents* (Figure 3.14). All the messages shown are independent of each other. For example, the $EXECUTETASK$ message is generated when the application controller has a task to execute. This message is then sent to the selected application agent which performs the desired operations, in this case executing the task locally. Figure 3.14 shows the common messages involved in it.



Message Description

1 ***GETLOAD***

Source: Application Controller
Destn.: Load Controller
Semantics:
Sent when the source needs to have updated load values from the destn.

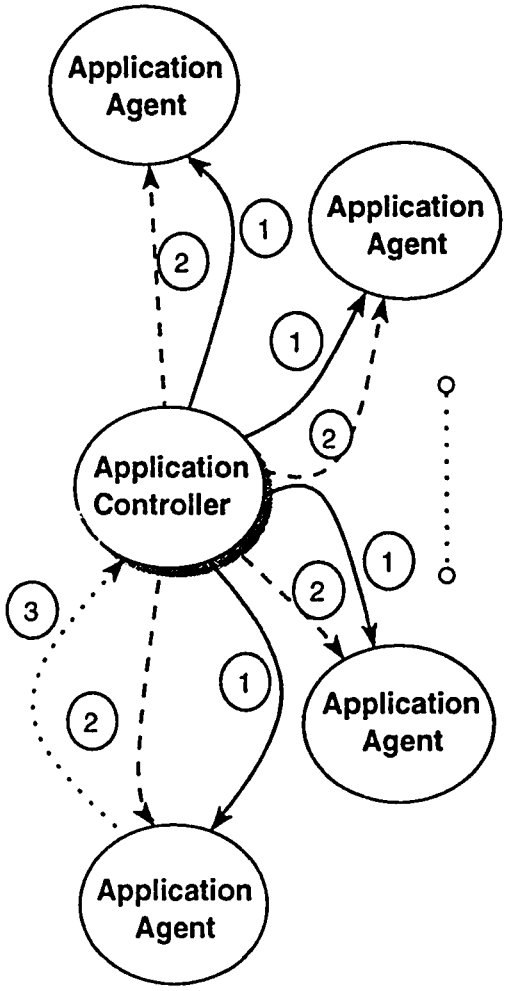
2 ***PARAMETERCTL***

Source: Application Controller
Destn.: Load Controller
Semantics:
Sent when the source needs to tune the parameters of the monitoring system.
Data:
- Identification of the Load Agents (could be broadcast or multicast
- Number of parameters to be modified
- List of Parameter Codes needed to be manipulated
- List of corresponding parameter values

3 ***CURRENTLOAD***

Source: Load Controller
Destn.: Application Controller
Semantics:
Sent either when the load refresh time has elapsed or as a response to the GETLOAD message
Data:
A List of the load averages of all the hosts in the *VM* in the order in specified by application controller in *IDSOFALLMONITORS* message during bootstrapping.

Figure 3.13: Messaging in the Application-Load Controller Protocol



Message Description

(1) ***EXECUTETASK***

Source: Application Controller
Destn: Selected Application Agent(s)
Semantics:
Execute a task on the host on which the destination is running.
Data:
- Name of the parallel application
- Name of the task
- Number of instances of the task desired
- Command line arguments, if any

(2) ***ENDTASK***

Source: Application Controller
Destn: Application Agent
Semantics:
The source sends this message to a destination to terminate a task . This could be the mechanism for providing dynamic task migration
Data:
- The application to which the task belonged
- Identification of the task
- Identification of the agent

(3) ***FINISHED***

Source: Application Agent
Destn: Application Controller
Semantics:
Sent by the source to indicate the termination of a task to the destination.
Data:
- The application to which the task belonged
- Identification of the task
- Identification of the agent

Figure 3.14: Application Controller-Agent Protocol

Chapter 4

LBFW - Implementation

4.1 Introduction

This chapter gives a detailed report of the implementation of *LBFW* based on the design described in chapter 3. *LBFW* was implemented as a distributed program running on a distributed programming support system called *Parallel Virtual Machine (PVM)*. *LBFW* in fact provides a wrap over the *PVM* by resulting in a customized distributed programming system for distributed and parallel applications, with load balancing.

4.1.1 Programming paradigms for Distributed and Parallel Applications

Any distributed or parallel application may be viewed as a collection of a number of tasks (processes) communicating with each other and working in a co-operative manner to solve the problem on hand. The heart of the gains involving any dis-

tributed or parallel application arise from the presence of *concurrency.* Concurrent programming is the art of constructing a program containing multiple processes that cooperate in solving a given problem.

Given a problem, decisions have to be made about what and how many tasks the application must contain? What is the best way of parallelizing it? How should the tasks communicate and synchronize? To answer such questions, guidelines are available in the form of programming paradigms for concurrent programming. Some of the commonly used paradigms are described:

Network of Filters Paradigm: A *filter* is a task whose output is a function of its input. In this paradigm, an application is visualized as a network in which vertices represent *filters* and links represent communication channels. Solutions to many problems can be implemented as a network of *filters.* Applications having a regular task graph, like binary trees, linear arrays, meshes etc. suit well to this paradigm.

Clients and Servers Paradigm: Clients send requests to servers which service these requests. Client processes send messages to a request channel and later receive results from a reply channel. This is by far the most common paradigm used in the field. Examples of clients and servers include file servers, graphics servers, authentication servers, Web servers and so on.

Heartbeat Paradigm: This paradigm is characterized by the following algorithm

```
Algorithm HeartBeat
  Initialize();
  Repeat
    Put out information from global variable on incident channels.
    Get information from neighbors into local variables.
    Compute new values using a function F.
    Evaluate a condition C to determine completion.
    If not C Then
      load global variables to be sent to the neighbors.
  Until C
End
```

Intuitively, this paradigm is more suited for the traditional multiprocessor systems.

Probe/Echo Paradigm: This is a concurrent programming analog of the depth first search of graphs. A *probe* is a message sent by one node to its successor, whereas an *echo* is a subsequent reply.

Supervisor/Workers Paradigm: This is a variant of the client-server paradigm. The supervisor breaks up a given problem into smaller sub-problems and puts them in an *outstanding queue*. The workers pick their work from the queue and put the results in a *result queue*. The supervisor makes sense out of the results in the result queue. This is a generalization of agenda-driven paradigm.

4.2 PVM - The working environment

The development of *PVM* was started in the summer of 1989 at Oak Ridge National Laboratories (ORNL) and has since then picked up as an important environment

for distributed programming. In fact some recent literature, for example [28] rate it as the *de facto* standard in the distributed computing world. Fundamental work in the development of *PVM* apart from ORNL was carried out at University of Tennessee Knoxville, Carnegie Mellon University, Pittsburgh Supercomputing center Pittsburgh and Emory University Atlanta [18].

4.2.1 Main Features

The *PVM* system essentially emulates a general purpose concurrent computing framework on a networked collection of independent computer systems. A system level process executes on each machine in a user configurable pool of processing elements, it is called the *PVM Daemon (pvmd).* The *pvmds* through co-operative distributed algorithms permit this collection of machines to be utilized as a coherent concurrent computing resource. Applications access this resource using a library of routines provided by it embedded in a procedural language as C or Fortran. Support is provided for process management, for communication via connection-less and connection oriented message passing, for synchronization based on barriers. Internally these library routines interact with the *PVM* service provider on each host i.e., the *pvmds.* Collectively, these daemons emulate a virtual parallel machine. A schematic display of the *PVM* system infrastructure is shown in Figure 4.1. It shows a *VM* consisting of three interconnected machines, a Sun, a NeXT and a DEC with a *pvmd* running on each. Four different user tasks running on it. Each task is communicating with the initializing *pvmd* and some of them are communicating with the others.

An *instance* of an application's task, realized as a process, is the unit of com-

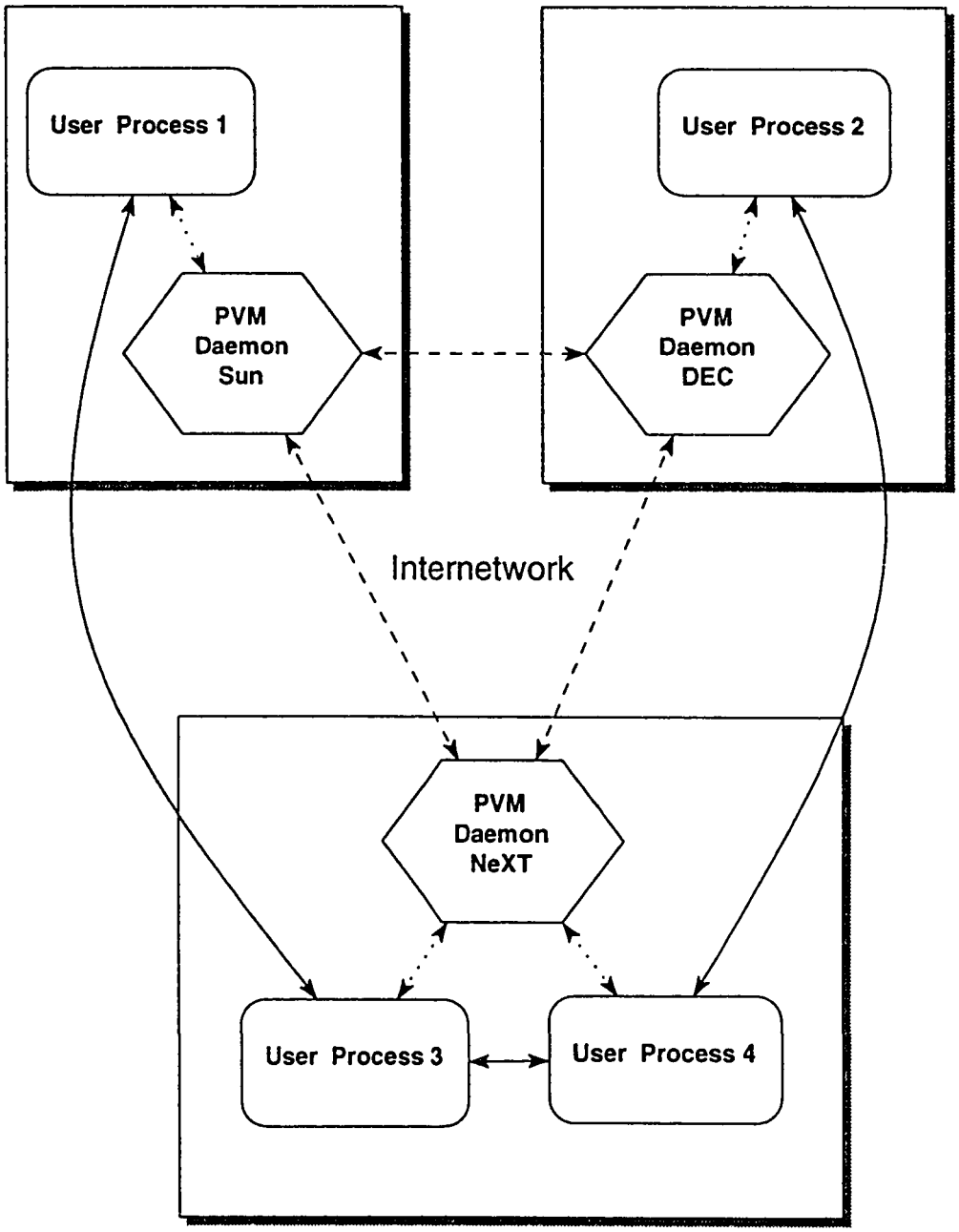


Figure 4.1: *PVM* Architectural Overview

putational abstraction in the *PVM* system. Each process is an executing instance of an application task. Therefore, an application can consist of several tasks, each of which may be dynamically manifested as multiple, instances that cooperate to solve a particular problem. Instances thus are independent sequential threads that may spawn or terminate other instances, synchronize with one or all, exchange data and do all that is needed to cooperatively solve the problem on hand. These factors were considered because of the design objective of supporting any type of parallel application. For example, Figure 4.2 shows a sample application having four components to solve some hypothetical problem. Each of the components represent a task instance. Further let *ComponentA* do the job of partitioning the user input and passing the suitable partitions over to the *ComponentB*. This takes its input and performs Matrix multiplication on it and returns the results to *ComponentD*. Similarly, *ComponentC* takes its set of partitions from *ComponentA*, performs Cholesky Factorization on it and returns the results to *ComponentD*.

Depending on the suitability of the algorithm of these components and the availability of hosts, one or more instances may be chosen. For example, the matrix multiplication of Component B, is more suitable for shared memory multiprocessor. The number of instances depend upon the size of the matrix under consideration.

Instances communicate via messages. Each message may contain multiple typed data areas. Messages are exchanged in terms of *packetized messages*. By default message exchange is asynchronous, in that a sending process may continue execution prior to physical message reception by the destination process. Blocking as well as non-blocking receive methods are provided. Synchronization is based on barriers. This support is needed for cooperative processing based applications where an in-

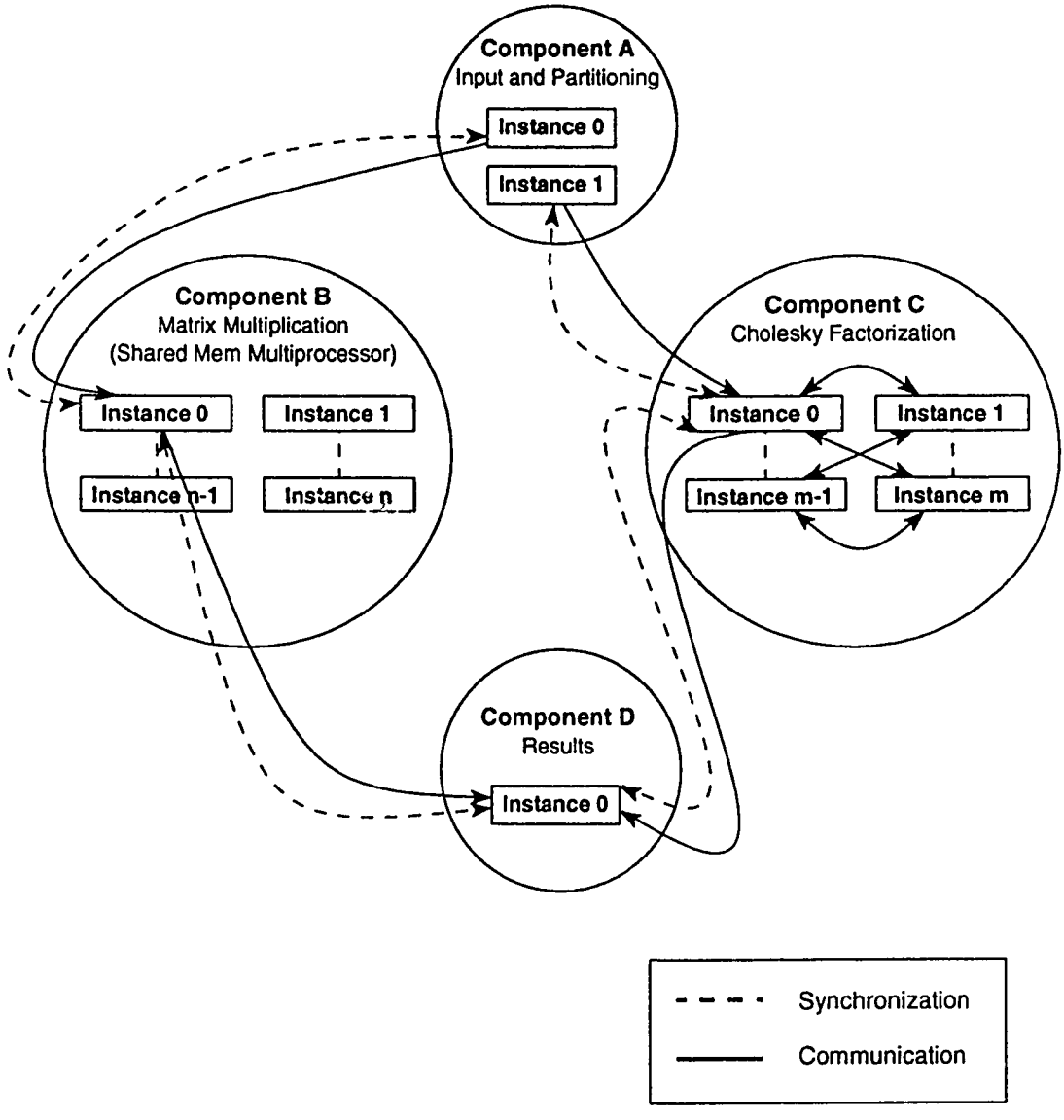


Figure 4.2: Example structure of *PVM* computing model.

stance after performing its execution must wait until the other processing elements catch up with it by sending appropriate messages.

4.2.2 Application Development

Applications utilizing the *PVM* infrastructure are coded in a manner similar to other distributed memory computing systems, e.g. the hypercube. However there are several significant differences:

Process Orientation: Under *PVM*, applications may use as many tasks or in other words processes as required. It could be very problem specific or it could be fairly general. This is unlike processor characteristics oriented parallel machines, instance to processor mapping need not be one-to-one. For example a multiprocessor having processors interconnected in the form of a tree is suitable for tree based applications only.

Multiple component execution: Support is provided for co-operative execution of multiple tasks. On traditional multiprocessor, cumbersome programming or manual intervention is required to accomplish this.

Exploiting environment heterogeneity: Since multifaceted virtual machines can be configured within the same framework, the potential for siting subtasks to the best suited architectures is significantly enhanced.

Logical interconnection: For both synchronization and communication, direct logical visibility is provided between arbitrary sets of task instances. For instance, in order to communicate with another task, a task just needs to know

the task identifier of the intended receiver rather than the socket port on which an accept is being waited for, what protocol is being used and so on.

Based on the above features it is evident that *PVM* provides an effective tool for distributed and parallel application development.Thus it was decided to use it as the medium of expression of the *LBFW* design.

4.3 Implementation Details

This section describes the details about *LBFW* implementation. The code of *LBFW* consists of the following programs:

- *application controller* This program is designed to include the load balancer module because of the close interaction between them. The clear interface between the load balancer and *LBFW* described in Section 3.4.1 is still maintained. A method of replacing a load balancer with any other load balancer was demonstrated by implementing three additional load balancing heuristics in addition to the one proposed. This is explained in Section 4.3.2,

- *history management* module,

- *library functions* to be provided to the user for aiding in application programming using *LBFW*,

- procedure to determine the virtual machine characteristics by parsing the specification files at *LBFW* startup,

- *application agent* module,

- *load controller* and
- *load agents.*

4.3.1 LBFW Library Interface

Figure 3.2 depicts a users view of *LBFW*. The user is assumed to code applications and run them under *LBFW*. One of the most difficult issues in distributed programming is message passing between the various components of a program. *LBFW* does not assume any knowledge of user programs, it is logical to assume the vice-versa. In order to hide *LBFW* implementation, functions are provided whose semantics are more related to the user domain of problem solving. These functions generate messages in suitable format for *LBFW* to understand. This set of library routines are implemented in the file *LibFuncs.c.*

These routines are provided to the user in the form of a library *lbfwlib.a.* This file contains functions which cater many common needs of a parallel or distributed system implementer. They include routines for:

- registering an application with *LBFW*,
- terminating a currently running application,
- obtaining the current load information about the *VM*. This service may not be used by all applications but is useful if the application is designed to do some useful action based on the *VM* load,
- executing tasks,
- sending and receiving data and control information and

- miscellaneous functions that are not related to any application but found to be useful while experimenting with *LBFW*.

An explanation of each routine is presented in the Appendix B.

4.3.2 Application Controller Implementation

Application controller code is present in the file *applicationController.c*. This file includes the startup module (Section 3.4.1.1) and the message interpreter module.

The message interpreter module contains the basic work-loop of the application controller (Figure 3.2). This section explains the implementation of each of the constituting routines in the form of pseudo-code.

SetupApplicationEnvironment: The execution of this routine is triggered by a message, generated by *LBFW* user interaction library, in response to call to *lbpvm_startApplication* routine. This generates a *READTASKFILE* message to the application controller and passes as arguments the name of the taskfile of user application and its user perceived name. The processing is shown in Algorithm 4.1.

As a response to this message, the message interpreter module then calls *parseTaskFile* function. This function takes the user specified arguments and returns a pointer to the application in the in-memory history. The technique used is to find out, based on the given user inputs, if the said application has ever been executed before. If so, then the history statistics are taken from the corresponding history file.

Algorithm 4.1 SetupApplicationEnvironment

Input: Name of taskfile, Task identifier of the user agent
Result: Pointer to In-memory history data structure
Processing:
Add Application Node To In-Memory History
While (NotEndOfFile (taskFile)) **do**
Read A Task Record
Update In-Memory Application History
Update Task Executable Size Sum
EndWhile

Search for a match in the History Name Mapping table

If (There is a match) **Then**
Setup a mapping between application name to the Unique Identifier
Initialize In-Memory history information from history files
Else
Generate new application identifier
Setup mapping between application name and the generated identifier
Initialize all values in In-Memory history
EndIf

SaveApplicationHistory: The reason for execution of this routine is a message generated in response to call to *lbpvm_EndApplication* routine. This generates an *ENDAPPLICATION* message to the application controller and passes as arguments the user perceived name of the application and the task-id of the user agent. The processing described in Algorithm 4.2.

Whenever this message arrives the application controller calls the *returnHistoryName* function to map the application name to a unique one. Then it calls the *freeAppNameMapping* function to free the In-Memory application name mapping. The function *saveApplicationHistory* then writes the contents of In-Memory history

information associated with the application in question to its corresponding history file. Finally, the *deleteFromAppList* function releases the memory being used by the data structures corresponding to the aforesaid application.

Algorithm 4.2 saveApplicationHistory

Input: User specified application name, User Agent task id
Output: Application history information saved to disk
Processing:
Get the mapped application name for the given name
Free the In-Memory application name mapping
Traverse the list associated with the application and write information to the corresponding history file
Delete In-Memory History information for the said application
EndProcessing

LoadBalancedSpawn: This routine actually implements the load balancing strategy in *LBFW*. It is called when the implementer uses the *executeTask LBFW* library call to have a task executed on the *VM*. This generates an $EXECUTETASK$ message to the application controller and passes as arguments the user agent task-id, the name of the application and the task name together with the number of instances of the task desired and also the commandline argument count and the arguments themselves. The processing is given in Algorithm 4.3:

Whenever such situation arises, the application controller first maps the user given application name to the system-wide unique name using the *returnHistoryName* function and then calls the *loadBalancedSpawn* routine. The parameters passed to this routine are the application name, the task name, the number of instances desired and command line arguments.

It then uses the *giveHostWithMinimumLoadFunction* routine for generating the

Spawn-List for the given execution request. This routine in turn calls the *returnExecTime* routine for finding the predicted execution time of the said task and the *commVolumeInBytes* function for finding the amount of data in number of bytes communicated between the said task and all the assigned ones so far. The function *commCost* then converts it into time in seconds based on the type of the *VM* currently used.

Algorithm 4.3 loadBalancedSpawn

Input: application name, task name, number of instances, argument count and argument list
Output: A load balanced mapping of the task to the *VM*
Processing:
Get the mapped application identifier for the given application name
If (application and task exist in In-Memory history) **Then**
For (each node of *VM*) **Do**
Evaluate the cost of running the task under consideration on that host (Algorithm 4.4)
EndFor
Else
return error cannot spawn
EndIf
Sort the cost array in increasing order
Calculate the ratio of distribution of tasks to the hosts
Generate the *Spawn-List* containing the number of instances of the task to be executed by each processor.
If (a host has to execute at least one instance) **Then**
Send *EXECUTETASK* message to the corresponding application agent
EndIf
EndProcessing

The function *sortIncreasingOrder* then sorts the costs associated with executing the subject task on each host of V, and the function *calculateRatios* calculates the ratios. Then the *Spawn-List* is generated by going through a simple while loop.

Algorithm 4.4 EvaluateOF

Input: A pointer to the task being requested to be executed, target host
Output: predicted cost if the task is executed on the said host
Processing:
Let s be a measure of *power* of the host
Let l be the normalized load on the host at that instant
Let x be the weighted average of completion times of the task with respect to the number of runs
Let c be the communication cost between all the currently executing tasks with which the subject task communicates.
Predicted Cost $= l * s * x + c$
EndProcessing

Therefore the basic job of the *giveHostWithMinimumLoadFunction* function is to generate the *Spawn-List.* When there is a need to incorporate a new load balancing strategy into *LBFW*, all that needs to be modified is this *giveHostWithMinimum-LoadFunction* function. The spawn list is then processed by the *loadBalancedSpawn* function again, which does the job of dispatching messages to the selected application agents to execute the given task.

4.3.2.1 VM Characteristics Module

This module in the current implementation is very simple but has the potential to be quite complex. It is important to chart out some characteristics of a *VM* to precisely distinguish between any two hosts of the *VM* if they are different. There are no fixed global benchmarks for this. This module is implemented so as to abstract all these changes and present a single value to the application controller which identifies the processing power of a host.

The function *giveHostCharacteristics* is the interface between the specification files (explained in chapter 3) and *LBFW*. It reads and understands them. The function *setupHostSpeedsArray* then uses some heuristic to convert these different data items into a meaningful number. In this implementation, 60% weight is given to collective MIPS of the processor, 25% to the main memory and 15% to the swap space available on the machine in question. This was kept constant for all the load balancing heuristic experimented.

4.3.3 History Management Module Implementation

It is the duty of this module to maintain and provide interface to the history information of applications to the other subsystems as and when it is needed. The main task of this module is to have a method of transparently converting a user given name of an application to a unique identifier to avoid conflicts. It allows the use of same history information of an application even if two different users use it.

The approach taken for tackling this problem is based on the following argument. Let the number of tasks of an application and the sum of size of the executable of each of its task for all possible architectures for which it is compiled, uniquely identify it. The possibility that this value is equal for two different applications is very low. If the application is compiled for a new architecture in addition to the current ones, then that would mean it is a new application. This could very much be the case if the characteristics of the processor are quite different. In the worst case, if it happens that there is a match, the available history information is loaded for making load balancing decision. The conversion from user specified application names to system-wide unique application identifiers is expressed in Algorithm 4.5.

Algorithm 4.5 ReturnUniqueApplicationName

Input: User perceived application name
Output: system-wide unique application name
Processing:
    For (task of the application) **Do**
        Sum up the size of executable of that task for each architecture
    EndFor
    Lookup On-Disk Index Table for a match
    If (There is a match) **Then**
        return the matched name
    Else
        Generate a new record for this application and put it in the On-Disk Index Table
    EndIf
EndProcessing

The On-disc index table is implemented as a simple file, each line is a record having three tab separated fields. The first field stores the number of tasks in the application, second stores the sum and the third field the corresponding unique name. The unique names used are *applicationx*, where x is a monotonically increasing integer. In the current implementation, the common repository shown in Figure 3.6 is the directory */pvm3/HistoryDirectory*. The index table is the file *historyTable* in that directory. The history information associated with each application is stored in the file *uniqueAppName.history*.

The function *sizeOf* takes as input the name of a task executable, and returns the sum of executable size of that task for all the architectures for which it is compiled. The function *searchHistoryTableFor* takes as input the number of tasks in the current application together with the sum of their executable and matches each record of *historyTable*. It returns the unique application name, if one exists, or

a null pointer otherwise. The function *createAppNameMapping* creates a new entry in the *historyFile*, if one is not found.

4.3.4 Application Agent Implementation

The file *applicationAgent.c* implements the application agent. It contains all modules explained in section 3.4.2.

The message interpreter module contains the basic work-loop of the application agent (Algorithm 3.3). This section explains each of the routines of the message interpreter.

ExecuteTask: Processing under this procedure takes place when the application controller sends an $EXECUTETASK$ message to the application agent, as described in Algorithm 4.6.

Algorithm 4.6 ExecuteTask

Input:taskName, applicationName, numberOfInstances, argc and argv
Output: task identifier of this task sent to the application controller
Processing:
Execute the task on the current host and note the execution time
Send the task identifier(s) back to the application controller
Add the executed tasks to the Task execution database
Request *PVM* to notify the task termination
EndProcessing

The function *lbpvm_spawn* executes the task. In addition this routine notes down the execution start time as a part of the history mechanism. This is because execution is done on the local host. Thus, the error associated with execution time measure-

ments is minimized. *PVM* as such does not provide any mechanism for finding the exact execution times of tasks running on it. The system call *getrusage* returns the resource utilization of a given process. There are, however, practical problem associated with using this system call. They arise due to its semantics. This call can be made by a process to find the resource usage of *its* children or *itself.*

The model of execution of a task in the *PVM* environment is as follows: Once a *pvmd* receives a request for executing a task, it *forks* a child and then *execs* the desired program in the child. Effectively, the *pvmd* is the parent of all the processes spawned. Thus, changes would have to be made to *PVM* code to get the exact execution time. Seemingly a disadvantage, it may be argued that it is in fact an advantage. The exact execution time would always remain the same for the task for a particular processor. This is definitely not the case, as it discounts other load on the machine, the network delays and all the other interferences.

This implementation, takes the wall clock time between the time of start of execution of the task and the time of execution of this task as its execution time. For obtaining the wall clock time the *gettimeofday* system call was used. The last step of the algorithm is a request to *PVM* to notify the application agent, as soon as the given task terminates.

UpdateProcessStructure: This routine becomes active as a result of termination of a task running on the same workstation as the application agent. This routine is activated when the message interpreter gets a $TASKTERMINATED$ message from *PVM*. The processing is presented in Algorithm 4.7.

Algorithm 4.7 UpdateProcessStructure

Input: Id of the terminated task
Output: None
Processing:
Get the time at which the message was received
Find the *task execution database* (Figure 3.10) record of the task
Update the task completion time field
Mark the task as terminated
EndProcessing

UpdateCommunicationVolume: This routine implements the trojan horse strategy for measuring the communication volume between tasks (section 3.4.2). It comes into picture when the application agent receives a *COMMUPDATE* message from an executing task, transparent to the application. The routine given in Algorithm 4.8 updates the communication volume in the task execution database (Figure 3.10).

Algorithm 4.8 UpdateCommunicationVolume

Input: Id of the reporting task, its name, application name, sender name, byte count
Output: None
Processing:
Find the *task execution database* (Figure 3.10) record of the task
If (Source Task has already sent some data) **Then**
Add up the byte count to the already received one
Else
Create a node for the communicating sender.
Initialize the byte count to the value reported.
EndIf
EndProcessing

SendCommInfoToAppController: Once a task terminates, the collected history statistics have to be transferred to the application controller. Thus logically, the list corresponding to the terminated task of Figure 3.10 is not needed at the application agent. But a problem might arise due to communication delays. All the messages containing the communication information may not have arrived at the application agent. Thus, by sending the statistics immediately, some information might be lost. An approach to solve this problem is that whenever a task terminates, a flag in the task node marks it as terminated. A time delay of a constant $GARBAGECOLLECTIOTIME$ imposed so that all the communication statistics may become available with the application controller. Once this time interval is elapsed, this routine is activated. It composes $FINISHED$ message with all the available statistics and sends it over to the application controller.

4.3.5 Load Controller Implementation

The file *centralLoadMonitor.c* implements the load controller (Algorithm 3.4), using a number of procedures.

GiveNormalizedLoadToAppController: This routine is called under two circumstances: for informing the application controller about the current VM load at regular time intervals, when there is an explicit request from the application controller asking it to supply the current VM load. Algorithm 4.9 explains its functionality.

Algorithm 4.9 GiveNormalizedLoadToAppController

Input:*GETLOAD* message
Output: *CURRENTLOAD* message
Processing:
  If (*GETLOAD* message is received OR
  It is time give updated load to application controller) **Then**
    send *CURRENTLOAD* message
  EndIf
EndProcessing

GetLoadFromLoadAgents: This routine is called once, when *LBFW* is in the startup process. It is also used when a specific request from the application controller for a fresh load collection from the *VM* is issued. The functionality is presented in Algorithm 4.10.

Algorithm 4.10 GetLoadFromLoadAgents

Input:*GIVELOAD* message
Output: *CURRENTLOAD* message after getting fresh load statistics
Processing:
  If (*GIVELOAD* message is received) **Then**
    send *GIVELOAD* message to all the load agents
    find out the load on the local host
    For (all load agents) **Do**
      receive *LOADSTATUS* message
    Done
    send *CURRENTLOAD* message
  EndIf
EndProcessing

RecieveLoadInfoFromLoadAgents: This routine is called whenever load agent sends its current load information to the load controller in the form of the

LOADSTATUS message. There is no periodic synchronization imposed in this scenario to avoid overhead. This can be justified as follows:

- All the load agents execute on different machines and are executed one after another. Due to the heterogenity involved, the slower host might become a bottleneck.
- No guarantee could be given about the times of clocks between the hosts to be the same.

Therefore waiting for *LOADSTATUS* message after every constant time is an inefficient approach. Even after assuming that there are no network and processor delays, there is no guarantee that all the messages will arrive in an acceptable span of time.

GetCurrentHostLoad: This routine is called by the load controller after a constant *UPDATETIME* amount of delay to ascertain the load on the control host. This function is common for both load controller and the load agents. In the current implementation, load is the average number of jobs in the run queue over the last one, five and fifteen minutes. These values were obtained by parsing the output of the uptime command.

4.3.6 Load agent Implementation

The file *distLoadMonitor.c* implements a typical load agent (Algorithm 3.5). The job of the load agent is to find the current load on the machine on which it is running and inform the load controller about it at appropriate times.

4.4 Overhead Assessment

This section presents the overheads that occur as a result of distributed programming. First the overheads associated with the socket interface are discussed then the overheads due to *PVM* followed by those due to *LBFW*.

4.4.1 Due to Socket Interface

This is the lowest level of support provided for application programmers intending to code distributed applications by a typical operating system like Unix. This interface has become a standard in contemporary computing. Sockets are analogous to mailboxes and telephones in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and the telephones allow access to the telephone system. Their position in the operating system is shown in Figure 4.3.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing connection, reading data, writing data and releasing the connection. Having a file descriptor is not enough, extra information (overhead) has to be maintained. For a Unix like system it is maintained in the *struct sockaddr* defined in *socket.h*.

The need for maintaining this extra information arises because of the differences between IO and networking when it comes to the actual hardware. In the case of IO, the device is usually located in the same computer. Whereas in the case of networking it may not be. The channel reliability in the case of IO and its speed are extremely high; it may not be true in the case if networking.

In order to provide transparency and a file like interface to the network, the

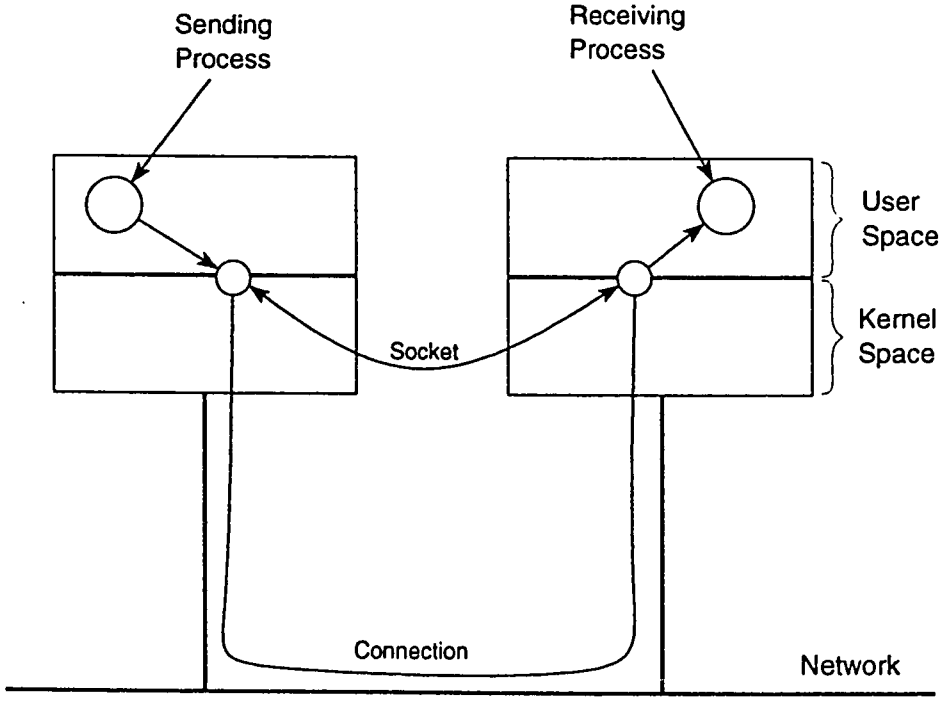


Figure 4.3: Use of sockets for Inter-process communication.

resultant overheads are unavoidable. These overheads are explained by the *types* of sockets being used. This inturn implies the type of protocol being used.

The most common protocols are referred to as the *TCP/IP* suite of protocols which provide the following socket types:

1. *Datagram:* These type of socket models potentially unreliable, connectionless packet communication.

2. *Stream Socket:* These model a reliable, connection oriented byte stream.

3. *Sequenced packet socket:* These model sequenced, reliable, unduplicated connection based communication that preserves message boundaries.

Having protocols to convert the unreliable communication medium, to a reliable one takes a lot of CPU as well as memory resources as overheads. They are adequately addressed in [29].

4.4.2 Due to PVM

PVM uses the socket interface for providing inter task communication. Therefore all the overheads associated with the socket interface are still valid for *PVM*. In addition to this, other overheads occur because of the following reasons:

- *PVM* provides *process orientated* communication(Section 4.2.2). In the case of *socket-port* model, a receiver socket should continuously listen on some well known port to accept connection from the sender. The sender in turn must know it. Thus, having inter-process communication this way (i.e. port oriented) between multiple senders and receivers will make the job of the user very difficult.

The methods to hide this is to maintain additional information about the task under execution. Details of these are found in [18]. Depending to whom the task is communicating with, associate the socket port of the sender to that of the receiver. Then a mapping scheme for doing the appropriate conversions.

- It is necessary to handle both connection-oriented and connection-less communication mechanisms in certain situations. For example, the protocol by which the *PVM* daemons communicate is more suitable for connectionless packet based communication. Problems with connection oriented stream communication are:

  1. The *VM* has to be scalable. Each TCP connection consumes a file descriptor whereas a single UDP socket can send and receive from any number of remote UDP sockets. Further, a *VM* composed of N hosts would need $N(N-1)/2$ connections, which would be too difficult to establish and more importantly maintain.

  2. Some minimum level of fault detection mechanism is necessary to be present. This is very difficult to achieve in TCP. It could be implemented using the TCP keep alive option. But a simpler approach of timeouts, at the *pvmd* level, works effectively in the case of UDP.

- The overhead of providing a logical messaging structure between communicating tasks. This means that two communicating tasks can concentrate on the semantics of the message rather than its syntax. This entails the use of additional buffers and routines to manipulate these buffers for fragmenting, encoding, sending and receiving the messages.

4.4.3 Due to LBFW

The *PVM* daemons *ping* each other regularly and keep themselves aware of the current *VM*. *LBFW* is implemented as an application on *PVM*, therefore the need of pinging different application components does not exist. As long as *LBFW* executes, apart from the standard *PVM* daemons, the following overheads are unavoidable.

The application controller is the compute server of *LBFW*. It is one of the most resource consuming components. It waits in a work loop waiting to get requests from the outside world and responding to them. It can get requests from application tasks currently running on *LBFW*, the various application agents and also the monitoring subsystem.

In implementing such a setup, an advantage provided by *PVM* for distributed programming, i.e. process orientation, now turns into a bottleneck. In traditional Unix-C programming, such situations (polling on multiple input sources) are realized using the *select* system call. This way the polling required is largely eliminated and thus not too much of additional load will be put on the machine because a workloop need not be continuously polled. In *PVM*, no such facility is provided. This is because, the descriptor being used for communication is *task identifier* and not *socket identifier*.

From a higher level of abstraction, a major source of overhead in *LBFW* is the need to maintain *application history*. Perhaps all the overhead associated in *LBFW* is in some way or another related to this issue. The driving factor behind this is to get the history information transparent to the user.

In order to get the information about the amount of data communication (in terms of bytes) that occurred between any task and any other task of a particular

application, the following method was used (Algorithm 4.8).

- Provide the user with *send* and *receive* primitives which not only do the job as intended by the user explicitly but also do the job needed to be done by *LBFW* behind the curtain. The *send* primitives transfer enough information about the sender that the *receive* primitive understands and updates the task execution database.

- Once the task terminates, the application agent which had executed the task, sends all the statistics it had gathered about the task in question to the application controller.

Chapter 5

Experimentation and Performance Analysis

In this chapter, the experimentation done using *LBFW* is discussed. In order to do so, a mechanism was developed for randomly generating the applications to be run on *LBFW*. The applications are run using four load balancing heuristics to demonstrate the feasibility of the history driven load balancing algorithm.

5.1 Writing a typical parallel application for LBFW (Parallel Selection)

The simple example chosen is the *selection* problem. *"Given a sequence S of n elements and an integer k, where $1 \leq k \leq n$, it is required to find the kth smallest element in S."* It arises in many applications in computer science and statistics. Algorithms 5.1 and 5.2 are taken from [30], which solves this problem on a shared mem-

ory SIMD model parallel computer and is adapted to distributed memory model. It is based on the sequential algorithm (Algorithm 5.2).

5.1.1 Algorithm

Algorithm 5.1 makes the following assumptions:

1. A sequence of integers $S = \{s_1, s_2, ..., s_n\}$ and an integer k, $1 \leq k \leq n$, are given, and it is required to determine the kth smallest element of S. This is the initial input to *ParallelSelect*.

2. The VM consists of N processors $P_1, P_2, ..., P_N$.

3. Each host of the VM has a capacity to store at least n/N elements in its local memory. At least one of the hosts has the capacity to store the entire data set in its memory.

Algorithm 5.1 ParallelSelect(S, k)

BeginProcessing
If $\mid S \mid \leq Q$ **Then**
P_1 uses at most $(Q + 1)$ comparisons to return kth element
Else
• Divide S into x sub-sequences S_i of equal length, where x is the number of sequential select processes used
• The sub-sequence S_i is assigned to the sequential task x_i.
EndIf

For i = 1 **to** x **do in parallel**
• Obtain the median M_i associated with sub-sequence S_i, using Sequential select
Store the values in a Median Array (M)
EndFor

Recursively call ParallelSelect to obtain the median of Median Array (M)

Divide the sequence S into three sub-sequences:

$L = \{s_i \in S : s_i < m\}$,
$E = \{s_i \in S : s_i = m\}$, and
$G = \{s_i \in S : s_i > m\}$.

If $\mid L \mid \geq k$ **Then**
    ParallelSelect(L, k)
ElseIf $\mid L \mid + \mid E \mid \geq k$ **Then**
        return m
    Else
        ParallelSelect$(G, k - \mid L \mid - \mid E \mid)$
    EndIf
EndIf
EndProcessing

Algorithm 5.2 works as follows:

1. A sequence of integers $S = \{s_1, s_2, ..., s_n\}$ and an integer k, $1 \leq k \leq n$, are given as initial input, and it is required to determine the kth smallest element of S.

2. At each stage of the recursion, a number of elements of S are discarded from further consideration and the kth smallest is found by the divide and conquer technique.

Algorithm 5.2 SequentialSelect(S, k)

BeginProcessing
If $\mid S \mid \leq Q$ **Then**
    Sort S and return kth element directly
Else

Divide S into $\mid S \mid /Q$ sub-sequences of Q elements each
(with up to $Q-1$ leftover elements)
EndIf

Sort each sub-sequence and determine its median.
Call *SequentialSelect* recursively to find m, the median of $\mid S \mid /Q$ medians found in the previous step.

Divide the sequence S into three sub-sequences S_1, S_2 and S_3 such that
$S_1 = \{s_i \in S : s_i < m\}$,
$S_2 = \{s_i \in S : s_i = m\}$, and
$S_3 = \{s_i \in S : s_i > m\}$.

If $\mid S_1 \mid \geq k$ **Then**
SequentialSelect(S_1, k)
ElseIf $\mid S_1 \mid + \mid S_2 \mid \geq k$ **Then**
return m
Else
SequentialSelect($S_3, k - \mid S_1 \mid - \mid S_2 \mid$)
EndIf
EndIf
EndProcessing

5.1.2 Implementation

Each of the Algorithms 5.1 and 5.2 form a task of the application. The user agent for this application is implemented as Program 5.1.

Program 5.1 *Code for the User Agent of Parallel Selection*

```
#include "myheaderfile.h"
main(argc, argv)
int argc;
char *argv[];
{
int  mytid,i, rv, *tids, N, pSelectTid;
int   ntasks, n, k;
char name[40], buffer[MAXBUF];
```

```
/*** Register this task with PVM. mytid is its unique task identifier ***/
   mytid = pvm_mytid();
/*** If unable to register with PVM  ***/
   if (mytid < 0 ) {
      printError (NOPVM, " Inside the User Agent ");
      exit(-10);
   }
/*** Register the application whose taskfile is in taskfile.selection and user agent tid
is mytid with LBFW ***/
   rv = lbpvm_startApplication(mytid, "taskfile.selection");
/*** Number of sequential select tasks to work in parallel ***/
   printf ("Enter the No. of Sequential Select Processes desired -> ");
   scanf ("%d", &N);
   tids = (int *) calloc(N, sizeof(int));
/*** Execute N instances of the sSelect task belonging to application selectionApp with
NULL arguments and return the task identifiers in the array tids ***/
   executeTask ("sSelect", "selectionApp", N, (char **)NULL, tids, YES);
/*** Execute 1 instances of the pSelect task belonging to application selectionApp with
NULL arguments and return the task identifier in the variable pSelectTid ***/
   executeTask ("pSelect", "selectionApp", 1, (char **)NULL, &pSelectTid, YES);
/***  Find out the value of n and k where n is the number of elements and k is the
kth smallest element to find. ***/
   printf ("Enter the Value of n and k respectively -> ");
   scanf ("%d %d", &n, &k);
/*** Inform LBFW that this task, userApp, belonging to application selectionApp will
communicate with other tasks, specified in taskfile  ***/
   lbpvm_initCommunication("selectionApp", "userApp");
/*** Initialize the send buffer for sending data ***/
   lbpvm_initsend(PvmDataDefault);
/*** Pack all the data to be sent ***/
   lbpvm_pkint (&mytid, 1, 1);
   lbpvm_pkint (&n, 1, 1);
   lbpvm_pkint (&k, 1, 1);
   lbpvm_pkint (&N, 1, 1);
   lbpvm_pkint (tids, N, 1);
/*** send the above packed data to pSelectTid task with a message tag = 800 ***/
   lbpvm_send(pSelectTid, 800); /*** Tids of the select tasks ***/
/*** Initiate a session of data receives from pSelectTid ***/
   lbpvm_StartRecv (pSelectTid, 802);
/*** Unpack the received information ***/
```

```
   lbpvm_upkstr (buffer);
/*** Terminate the active data receive session  ***/
   lbpvm_EndRecv();
   printf ("%s \n", buffer);
/** Kill the sSelect Tasks  **/
   for (i = 0; i < N; i++)
      pvm_kill (tids[i]);
   scanf ("%d", &n);
/*** Indicate to LBFW of logical termination of the application ***/
   lbpvm_endApplication("selectionApp");
}
```

The call to the routine *pvm_mytid* registers this program with *PVM*. Each task running under the *PVM* has a unique *task identifier(tid).* This call returns the *tid* of the task. The routine *lbpvm_startApplication* indicates to the application controller that the application described in the taskfile *taskfile.selection* will be executed by the user agent whose *tid* is *mytid.* In other words, it registers the applications with *LBFW.* This results in setting up the application execution environment, i.e in-memory history database.

The contents of the *taskfile.selection* are as follows:

```
selectionApp

pSelect userApp sSelect end

sSelect pSelect end
```

The first line indicates that the name of the application, as the user sees it, *selectionApp.* The second line of *taskfile.selection* indicates that the task *pSelect* may receive data from *userApp* and/or *sSelect* tasks. The keyword *end* indicates the end of each logical record. The third line indicates that the task of type *sSelect* receives data from *pSelect.* The task *pSelect* implements the parallel selection algorithm, *sSelect* implements the sequential select algorithm and *userApp* is the compiled for

the user agent shown in Program 5.1.

Call to routine *executeTask* of Program 5.1 requests the execution of the task passed in the arguments. For example, the call *executeTask ("sSelect", "selectionApp", N, (char **)NULL, tids, YES);* requests the execution of the task *sSelect* belonging to the application *selectionApp* and execute N instances of the task with no command line arguments and return the *tids* in the integer array *tids*, if required. The flag YES indicates that the task ids are to be returned. The second call to *executeTask* requests execution of one instance the *pSelect* task of *selectionApp*. Note that *pSelect* partitions the data, distributes appropriate partitions to *sSelect* tasks and compiles the results.

Call to routine *lbpvm_initCommunication* in Program 5.1 indicates to *LBFW* that the task *userApp* of application *selectionApp* wishes to enter into communication with *pSelect* as shown in the taskfile. The routines *lbpvm_initsend* and *lbpvm_pk** provide a method of initializing the send buffer and packing data into it, respectively.

The *lbpvm_send* routine sends the active sends the data packed in the active send buffer to the *pSelect(Tid)* task with a message tag of 800. Routines *lbpvm_startRecv* and *lbpvm_EndRecv* between them embed all the data receive instructions from *pSelect*. The data is retrieved using the *lbpvm_upk** instructions. In the above case, a string is received in *buffer* as the result.

5.2 Load Balancing Heuristics

This section describes in detail the various load balancing heuristics implemented for comparing their performance against *lbHeuristic*. All heuristics are compared

for the same application.

Random Heuristic: This does not take any task, application, or system information into account for the purpose of scheduling. For each request to execute a task, all it does is to generate a random number between 0 and $N-1$ where N is the number of the hosts in the system. The task is then sent to the selected host. It is implemented as Algorithm 5.3.

Algorithm 5.3 RandomHeuristic

Inputs: application name, task name, number of instances, arguments
Outputs: Spawn-List (Section 3.4.1.5)
Processing
    While (number of desired instances are not executed) **Do**
        Generate randomly, a host on which tasks have to be allocated
        Generate randomly how many instances are to be allocated
        Update the Spawn-List
    EndWhile
    If (a host has to execute at least one instance) **Then**
        Send *EXECUTETASK* message to the corresponding application agent
    EndIf
EndProcessing

Implementing Algorithm 5.3 in *LBFW* needs change only to the *giveHostWithMinimumLoadFunction* function of the program *applicationController.c.*

Fixed Assignment Heuristic: This heuristic also does not take any application information into consideration. The algorithm uses equal distribution of application load between the available hosts. The strategy applied for implementing this is that an instance of each task execution request is given to the available hosts in cyclic order. Thus if m task execution requests arrive, and $m = k * n$

where n is the number of hosts in VM and k is a constant, then every host gets k task instances to execute. Algorithm 5.4 gives the change to be done to the function *giveHostWithMinimumLoadFunction.*

Algorithm 5.4 FixedAssignmentHeuristic

Inputs: application name, task name, number of instances, arguments
Outputs: Spawn-List (Section 3.4.1.5)
Processing
Let *currentHost* denote the host on which next task instances has to execute
While (number of desired instances are not executed) **Do**
Allocate next task instance on *currentHost*
Update *currentHost* in a cyclic manner between 0 and $N-1$
(Where N is the number of hosts in the VM)
Update the Spawn-List
EndWhile
If (a host has to execute at least one instance) **Then**
Send $EXECUTETASK$ message to the corresponding application agent
EndIf
EndProcessing

Opposite Heuristic: This heuristic behaves in an exactly opposite way to the proposed heuristic (3.4.1.5). The proposed heuristic (*lbHeuristic*) does load balancing on the basis of proportionate distribution of workload based on the evaluated objective function as explained in algorithms 4.3 and 4.4 respectively. The working of *lbHeuristic* is reversed by making inversely proportional distribution of workload. Thus the host which is found to be the most unsuitable for the execution of the task in question is assigned the responsibility of executing it. The only difference this would need from *lbHeuristic* is that instead of sorting the *cost array* in *increasing* order, it is done in *decreasing*

order Algorithm 4.3.

5.3 Experiment Design for Performance Analysis

A large number of real parallel applications exist. Looking into them is out of the scope of this work. Still we wanted to pass *LBFW* through rigorous testing.

Any distributed or parallel application can be modelled by a graph where the nodes represent tasks of the application, links between nodes denote communication between tasks and the direction on the link expresses the direction of data transfer. For all parallel applications, the above features are common. Therefore from this point of view, applications differ as follows:

- *Distribution of task types is random:* What tasks do in an application is directly related to the algorithm of the application. In a general sense, each task has to do either some computation, indulge in input and output, communicate with other tasks or a combination of the three.

- *Random Communication Pattern:* The way the tasks communicate, again, is completely dependent on the application. Some of them may communicate with more than one task while the others may not communicate with any.

- *Random Load Distribution:* The amount of work done by each task, be it in terms of computation or communication or input output. This again is an application dependent information.

Hence we developed the idea of a *Generic Pseudo Parallel Application (GPPA)*. A

GPPA itself is a distributed or parallel application, except that it does not do any useful job. It simulates different parallel applications having random task types, communication pattern and load distribution.

All the tasks of a *GPPA* are basically identical. They differ depending on the *control parameters.* The control parameters are defined as those parameters which characterize the way in which a task of a *GPPA* behaves. For example, a control parameters of a task to indicate that the task has high granularity could be that the task is *highly CPU* intensive. This notion of *highly* is quantified by a value which indicates how many times a calculation is performed to simulate this task. They are discussed in detail in Section 5.4. Algorithm 5.5 gives the basic function of each task.

Algorithm 5.5 genericTask

```
Begin genericTask
   receiveParameters;
   For n times Do
      performProcessing;
      If (there are any out − edges) Then
        sendOutputsToTasksOnOutEdges;
      Endif
      If (there are any in − edges) Then
        getInputsFromTasksOnInEdges;
      EndIf
   EndFor
End genericTask
```

As soon as a task starts execution, it waits to receive its *control parameters* from the *user agent*, and receives them in the *receiveParameters()* routine. Each task of a tree type *GPPA* is characterized by the following control parameters :

- The *type* of the task. This can be either a CPU, IO or COMmunication oriented. It determines the type of processing the task does in the *performProcessing()* routine.

- The *granularity* of the task. This determines the how much work the task has to do. If the granularity is high, it means that the number of times the operations have to be performed is comparatively large.

- The *type* and *granularity* of tasks on its incoming edges. This is necessary as the receiving task must know what kind of data the sender is going to send so that appropriate receive mechanism can be activated.

If the task is CPU oriented, then it performs some number crunching operations. Similarly the IO oriented tasks performs a series of file operations on a file, whereas the COMmunication oriented task send/receive messages to/from tasks on its outgoing/incoming edges.

The routine *performProcessing()* simulates the actual work done by the task and routine *sendOutputsToTasksOnOutEdges()* sends the result of processing to the tasks on its out-edges. The routine *getInputsFromTasksOnInEdges()* receives the inputs to a task. The rest of the algorithm is self explanatory.

5.4 GPPA Generation & Characterization

In order to remove the errors due to human bias, an automated scheme for generation *GPPA* along with their controlling parameters was devised. The random number generation utility used was the Unix *random*() function with a seed value as the number of seconds that have elapsed from January 1, 1970 to the current time.

Random() uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31} - 1$. The period of this random number generator is very large, approximately $16 * (2^{31} - 1)$ [31].

The application generation and characterization is done using two programs. The input to the first is the number of tasks of the *GPPA* and the range in which the granularity of the tasks of the application (minimum and maximum). This results in application *specification* file which contains, for each task, a randomly assigned type and load value.

The purpose of the second program is to impose interconnection between the tasks. It takes the number of tasks as input and generates the application *taskfile*, which contains the application task graph. This is a replaceable program which could either generate regular application task graphs or irregular. In the experiments carried out, apart from applications having random task graphs, task graphs of binary tree and mesh were generated.

5.5 Experimental Setup

This section describes the experimental setup used. All the experiments were done on the *CCSE Network@KFUPM*. The network runs on a 10Mbps Ethernet. From a variety of machines, a number of Sun workstations and NeXT LanStations were used. The experiments were carried out at different times and no assumption was made regarding the current status of the Network load, System or Workstation load. A number of different sets of random *GPPA* were generated for experimentation. Here we describe the results of four of the test cases. They are named Test1 thru

Test4.

For each test case, the following experiments were performed:

- The corresponding *GPPA* was executed fourteen times one after the other under four load balancing heuristics, incorporated in *LBFW*, in the order:

  1. *lbHeuristic*
  2. random assingment heuristic
  3. fixed assignment heuristic and
  4. a heuristic opposite to *lbHeuristic*

- Between any two executions of a set, a gap of two minutes was given.
- The execution time of each task is recorded for every execution.
- Total execution cost of each application is calculated for every run. This is done by adding up the execution time of each task and numbering it *run-wise.*
- Average execution time is calculated per task over all the runs. This is calculated by adding the execution time of each task and dividing it over the number of runs.
- Application completion times were found wherever possible. This is defined as the time it took the last task to complete.[1]

[1]All the time measurements are in terms of seconds

5.6 Experimental Results and Discussion

This section explains the results obtained for every test case one after another. The first class of tests is based on binary tree type task interconnection (Test1, Test2, Test3). The second class is based on mesh type task interconnection (Test4).

5.6.1 Test 1

Figure 5.1 shows the application task graph, binary tree in nature, having fifteen tasks. Tabular representation of the same is shown in Table5.1. In this and all *GPPAs* having binary tree form of task graph, i.e. Test2 and Test3, parent child relationships can be read from the specification tables using the following method. For each task name t_i, if a task $t_{i/2}$ exists then it is the parent of t_i. Similarly, if a task t_j exists where $j = 2i$, for a task t_i, then t_j is the *left child* of t_i. If $j = (2i+1)$ then t_j is the *right child* of t_i.

.

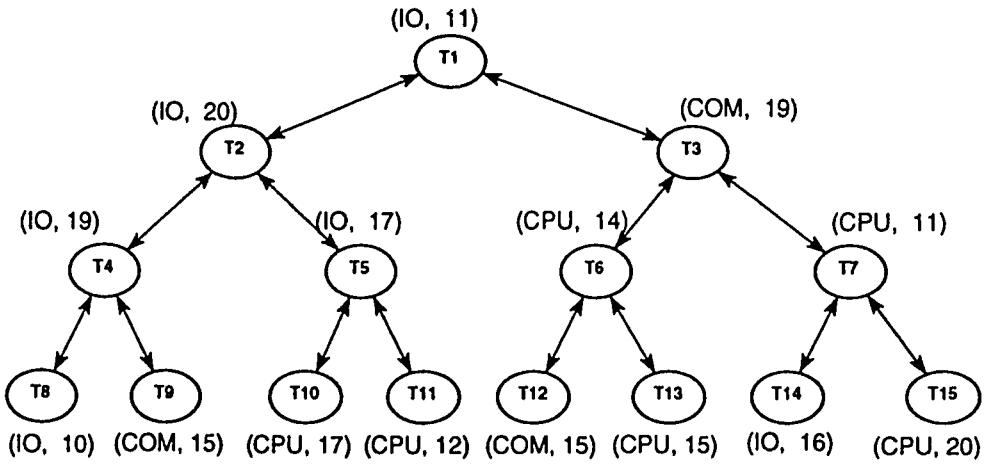


Figure 5.1: **Test1:** Task graph

Task	T1	T2	T3	T4	T5	T6	T7	T8
Type	IO	IO	COM	IO	IO	CPU	CPU	IO
Granularity	11	20	19	19	17	14	11	10
Task	T9	T10	T11	T12	T13	T14	T15	-
Type	COM	CPU	CPU	COM	CPU	IO	CPU	-
Granularity	15	17	12	15	15	16	20	-

Table 5.1: **Test1:** Specification

The *GPPA* of Table 5.1 shows that the granularity range of tasks is between ten and twenty. There are six *CPU* type tasks, six *IO* and three *COM*. The average granularity of tasks was 15.4 and the standard deviation between them was 3.22.

5.6.1.1 Total Execution Cost Measurement

Figure 5.1 is one of the demanding *GPPAs* experimented, having a relatively higher granularity. The average of total execution cost of *lbHeuristic* is much superior to the other heuristics as is evident from Table 5.2. The graph in Figure 5.2 shows that fixed assignment heuristic performs consistently better performance than the random assignment heuristic. The performance of *lbHeuristic* shows a sharp increase on the second run. This could be because of a bad estimate of the execution requirements based on the only previous history information available.

5.6.1.2 Average Execution Time Measurement

The graph in Figure 5.3 shows the results of the *GPPA* defined in Table 5.1. *lbHeuristic* gives almost twice as better results than the closest heuristics.

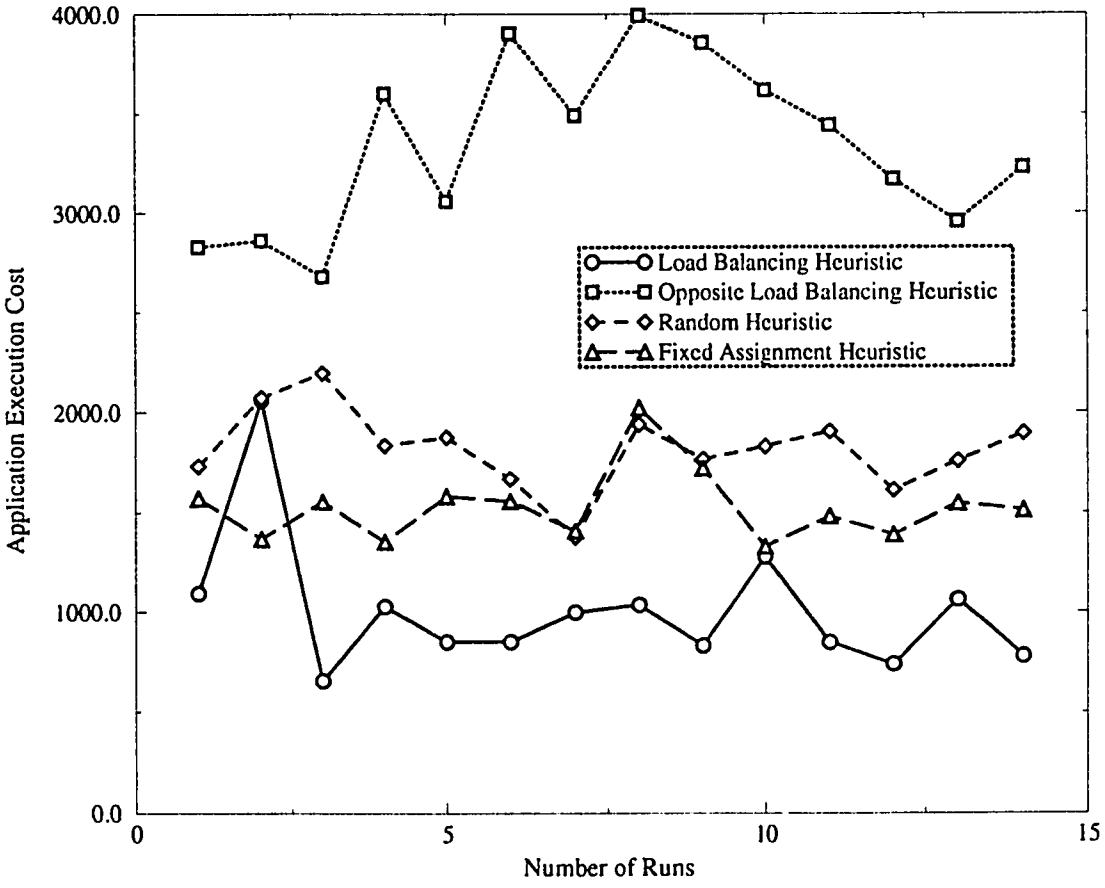


Figure 5.2: **Test1:** Total Execution Cost Comparison

Heuristic	Average Total Cost Per Run
LB	1006.5
OPPLB	3333.21
RANDOM	1820.5
Fixed	1529

Table 5.2: **Test1:** Average of Total Execution Cost

5.6.1.3 Application Completion Time Measurement

Figure 5.4 represents the comparison of completion times. The average completion time of *lbHeuristic*, follows very closely the total execution cost graph, is about 1.5 times better than the nearest heuristic (Table5.3).

Heuristic	Completion Time
LB	96.07
OPPLB	276.64
RANDOM	167
Fixed	135.42

Table 5.3: **Test1:** Average Completion Time measurements

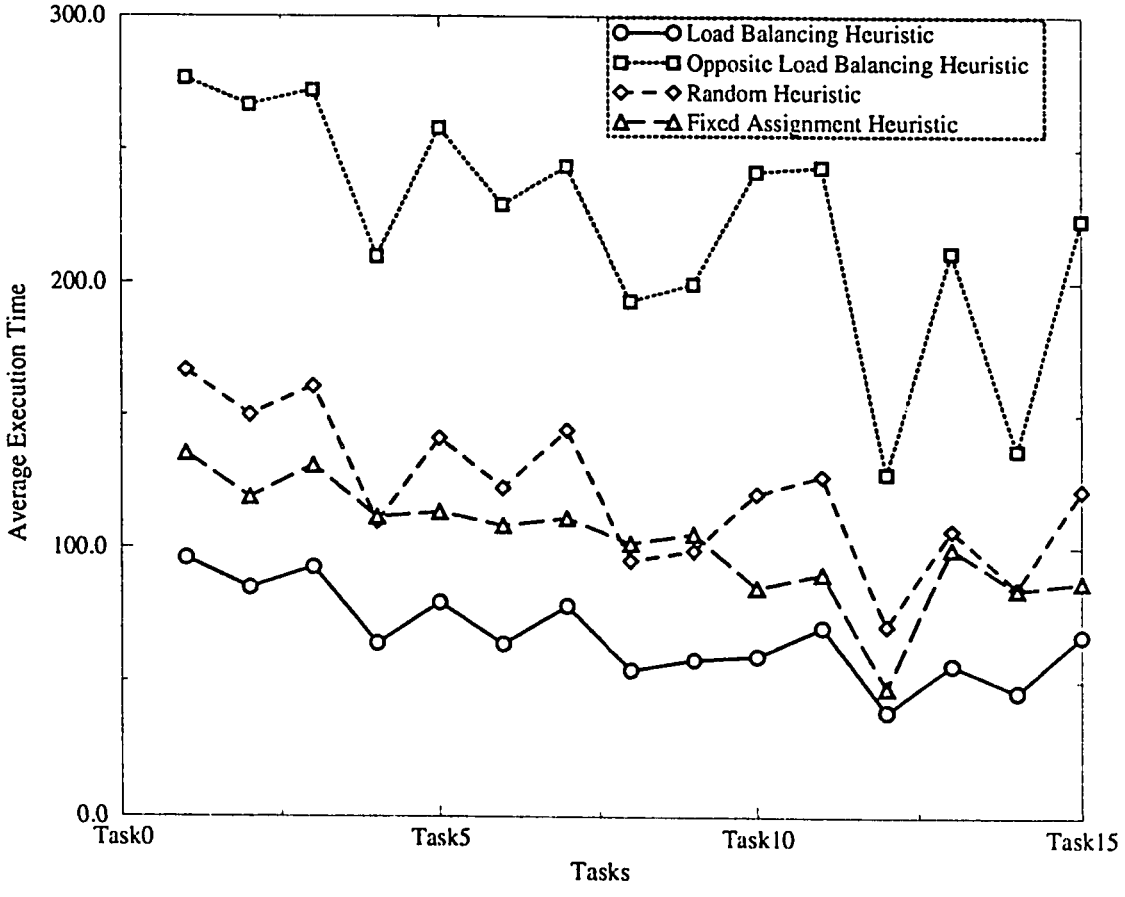


Figure 5.3: **Test1:** Average Task Execution Time Comparison

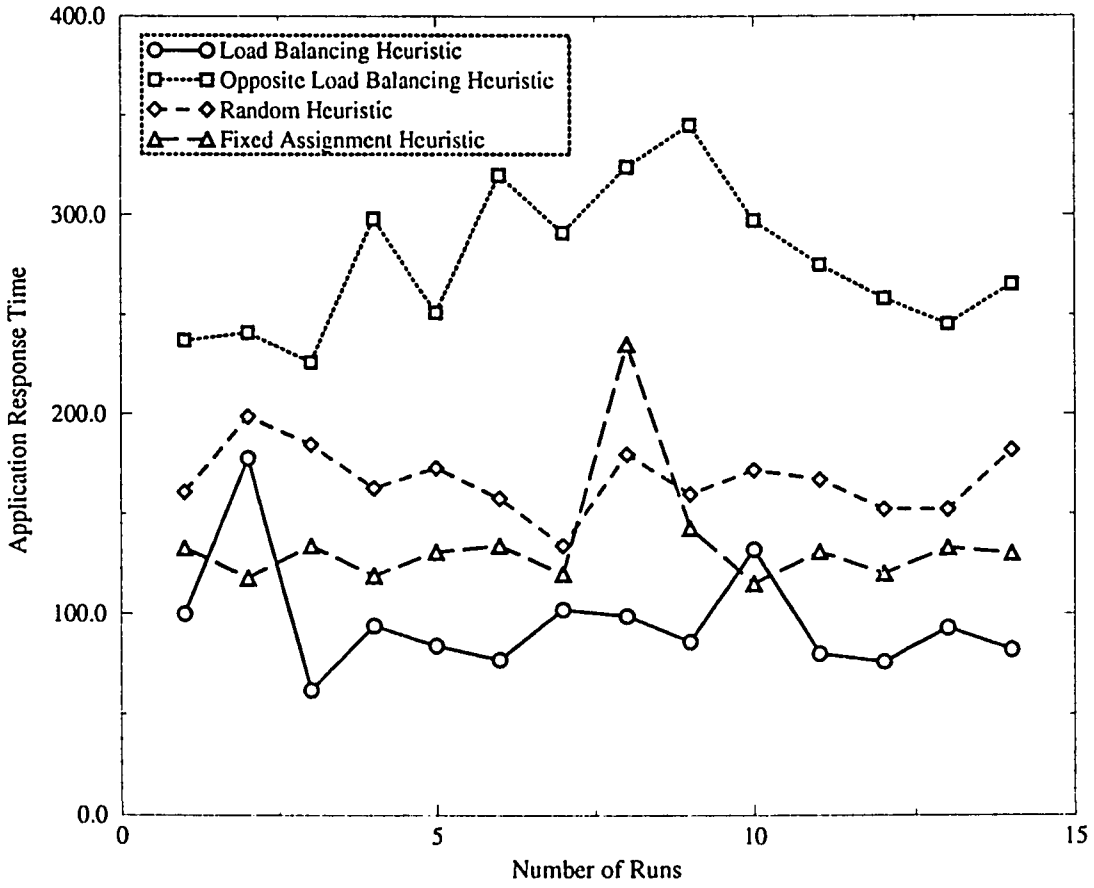


Figure 5.4: **Test1:** Application Completion Time Comparison

5.6.2 Test 2

The *GPPA* of Table 5.4 is derived from the *GPPA* of Table5.1. The objective is to test the performance of *LBFW* under simulated condition of having *CPU* type tasks of relatively high granularity. Thus all the tasks are of *CPU* type, with the granularity distribution being same as in Table 5.1

Task	t1	t2	t3	t4	t5	t6	t7	t8
Type	CPU	CPU	CPU	CPU	CPU	CPU	CPU	CPU
Granularity	11	20	19	19	17	14	11	10
Task	t9	t10	t11	t12	t13	t14	t15	-
Type	CPU	CPU	CPU	CPU	CPU	CPU	CPU	-
Granularity	15	17	12	15	15	16	20	-

Table 5.4: **Test2:** Specification

5.6.2.1 Total Execution Cost Measurement

A performance similar to Test1 is observed in graph of Figure 5.5. Table 5.5 presents the average completion time of all the tasks, where *lbHeuristic* shows over two times better performance than the second best random heuristic. During experimentation with *random* and *fixed assignment* heuristics, due problems with the network, all the 14 runs could not be completed. Nevertheless, the performance showed consistently inferior results when compared with *lbHeuristic*.

5.6.2.2 Average Execution Time Measurement

The graph in Figure 5.6 shows the comparison of average execution time of each task of Test2 (Table 5.4) over 14 runs. In this case also, the *lbHeuristic* gives much

better results than the other heuristics compared.

Heuristic	Completion Time
LB	1629.92
OPPLB	6729.35
RANDOM	3751.1
Fixed	4801.5

Table 5.5: **Test2:** Average of Total Execution Cost

5.6.2.3 Application Completion Time Measurement

The graph in Figure 5.7 shows application cost completion time results in the case of Test2. Table 5.6 gives the average completion time of the application over all runs. The performance of *lbHeuristic* is over two times better than the the next best heuristic.

Heuristic	Completion Time
LB	150
OPPLB	529.64
RANDOM	305.3
Fixed	395.6

Table 5.6: **Test2:** Average Completion Time measurements

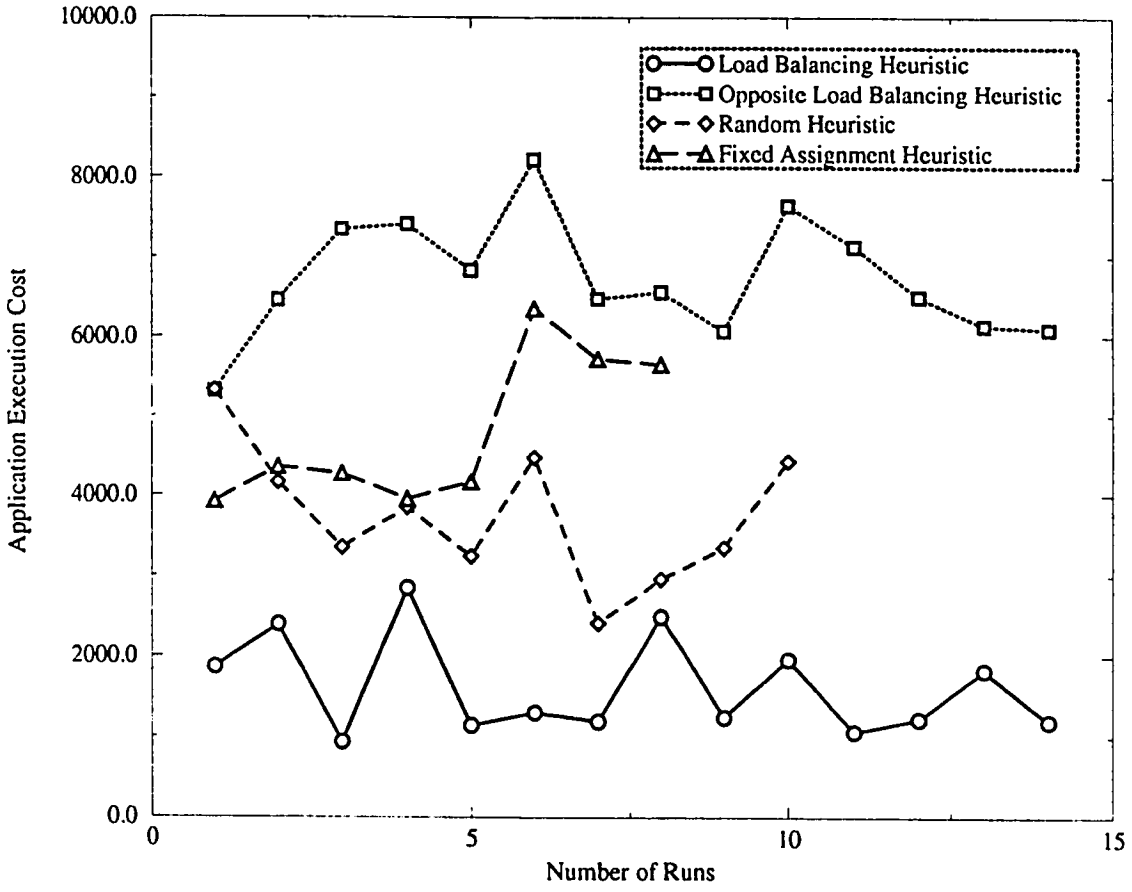


Figure 5.5: **Test2:** Total Execution Cost Comparison

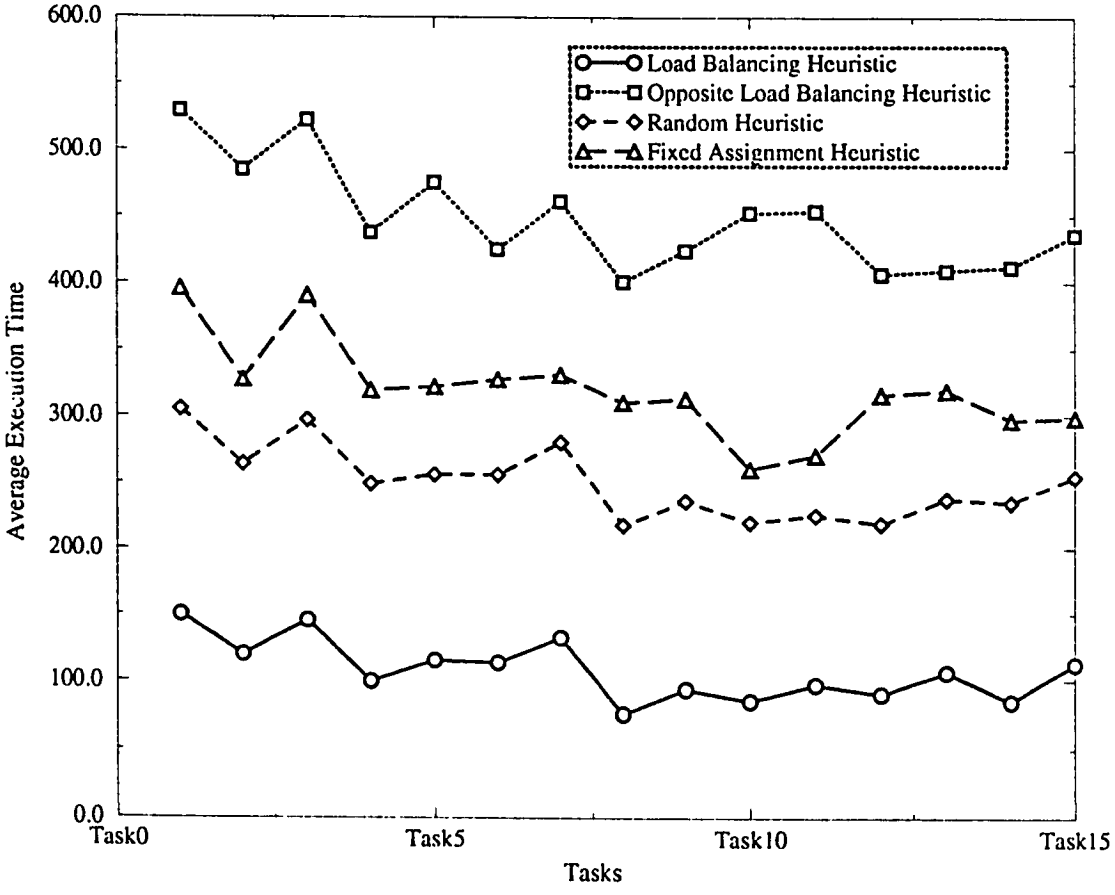


Figure 5.6: **Test2:** Average Task Execution Time Comparison

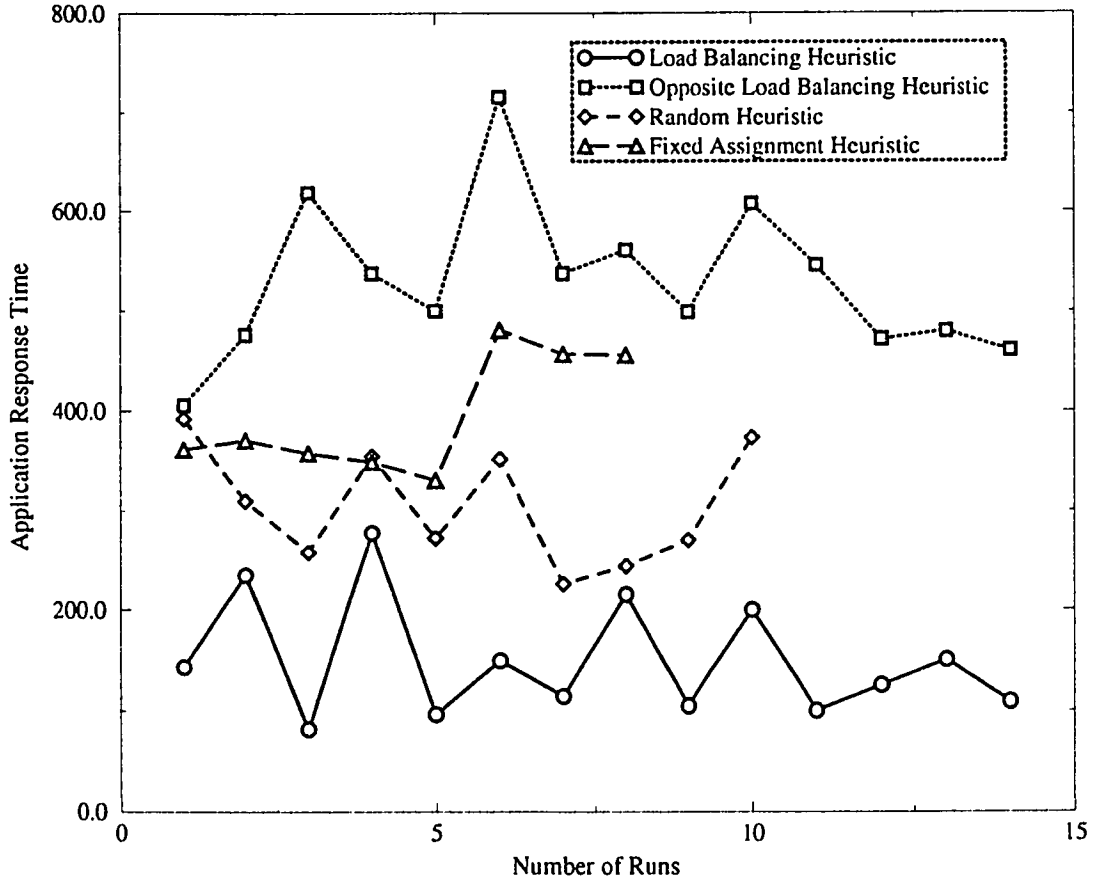


Figure 5.7: **Test2:** Application Completion Time Comparison

5.6.3 Test 3

The *GPPA* in Table5.7 is also a fifteen task binary tree type application. It contains four *CPU*, four *IO* and the seven *COM* tasks. This *GPPA* has predominantly communication oriented tasks. The range of task granularities was from one to ten. The standard deviation in the task granularity is 2.82 and mean is 5.6.

Task	t1	t2	t3	t4	t5	t6	t7	t8
Type	COM	IO	COM	IO	COM	COM	CPU	COM
Granularity	10	1	2	5	3	6	7	6
Task	t9	t10	t11	t12	t13	t14	t15	-
Type	CPU	IO	IO	CPU	COM	COM	CPU	-
Granularity	3	7	9	5	9	2	9	-

Table 5.7: **Test3:** Specification

5.6.3.1 Total Execution Cost Measurement

Graph in Figure 5.8 compares the total execution cost of the *GPPA* specified in Table 5.7. The results obtained from this experiment are very interesting because they seem to be in accordance with the fact that communication cost is an important bottleneck. The *fixed assignment* heuristic performs almost same as *LBFW*. There could be many additional reasons:

- A sudden increase in the *VM* load (system as well as network load).
- It may just be a coincidence where mapping chosen by the fixed assignment algorithm happened to be as good as the one chosen by *lbHeuristic* because of the problem overheads.

- *lbHeuristic* is history based, which requires additional processing. In this case, since the majority of tasks have lower granularity, the history management overhead could have been dominant.

Table 5.8 presents the average completion time of all the tasks.

Heuristic	Completion Time
LB	498
OPPLB	1180
RANDOM	808
Fixed	548

Table 5.8: **Test3:** Average of Total Execution Cost

5.6.3.2 Average Execution Time Measurement

The graph in Figure 5.9 shows the average execution time results of the *GPPA* specified in Table5.7. Again, as discussed in Section 5.6.3.1, the performance of *lbHeuristic* is almost same as the fixed assignment heuristic.

5.6.3.3 Application Completion Time Measurement

The graph in Figure 5.10 shows the application completion time results for the *GPPA* specified in Table5.7. As seen Sections 5.6.3.1 and 5.6.3.2, the performance of *lbHeuristic* was almost same as that of *fixedAssignment* heuristic. Table 5.9 gives the average completion time of the application over all runs.

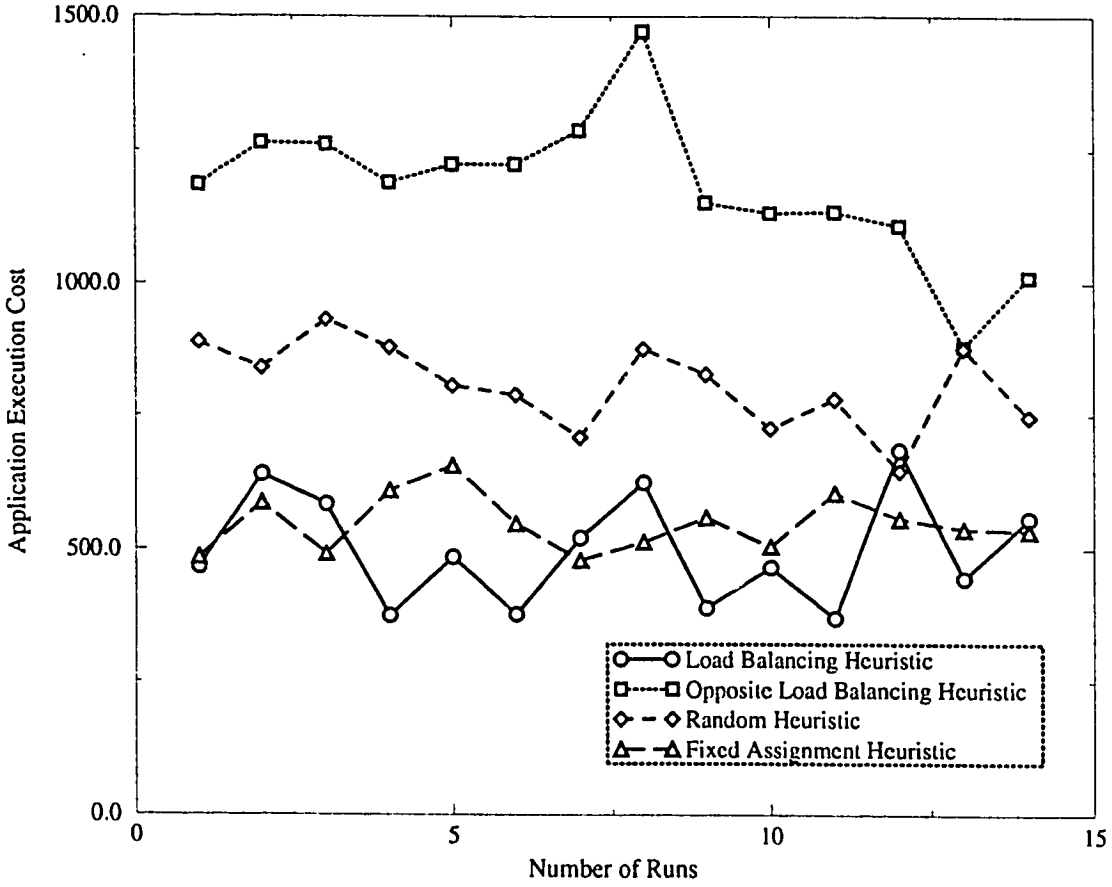


Figure 5.8: **Test3:** Total Execution Cost Comparison

Heuristic	Completion Time
LB	58.92
OPPLB	124.14
RANDOM	93.35
Fixed	59.71

Table 5.9: **Test3:** Average Completion Time measurements

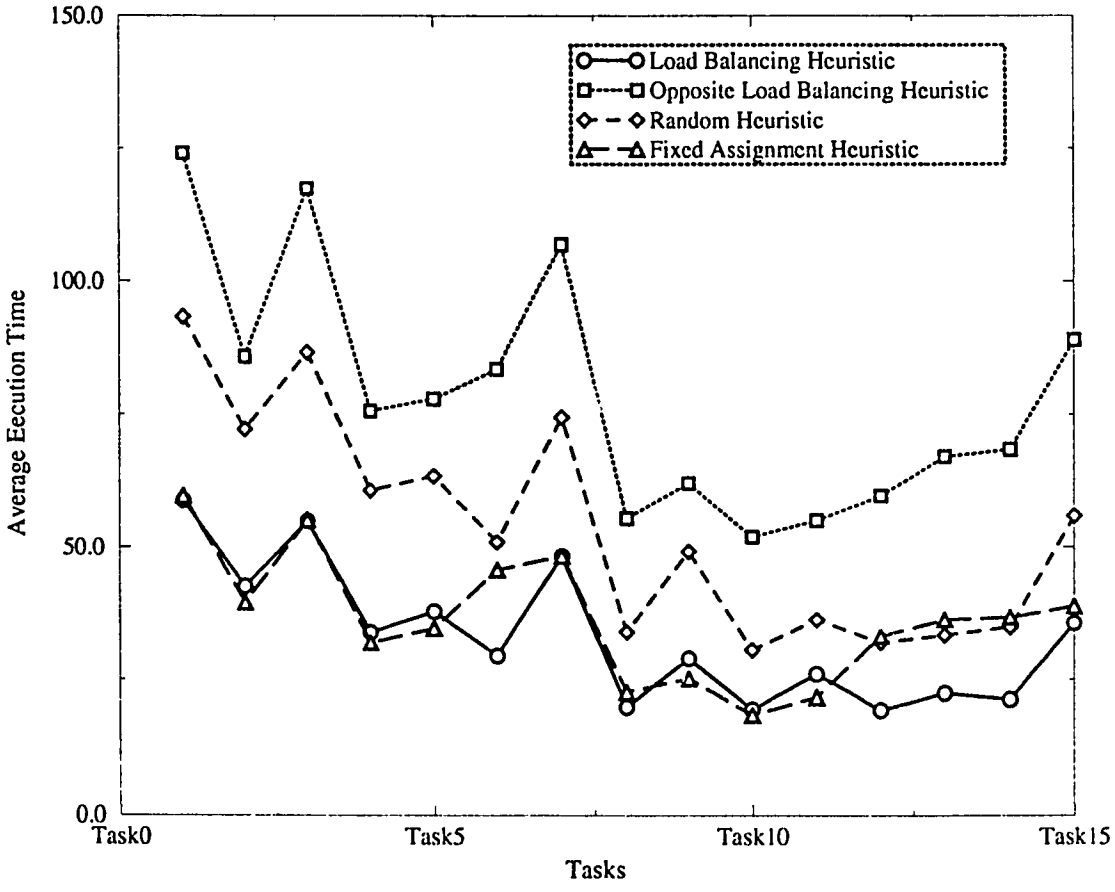


Figure 5.9: **Test3:** Average Task Execution Time Comparison

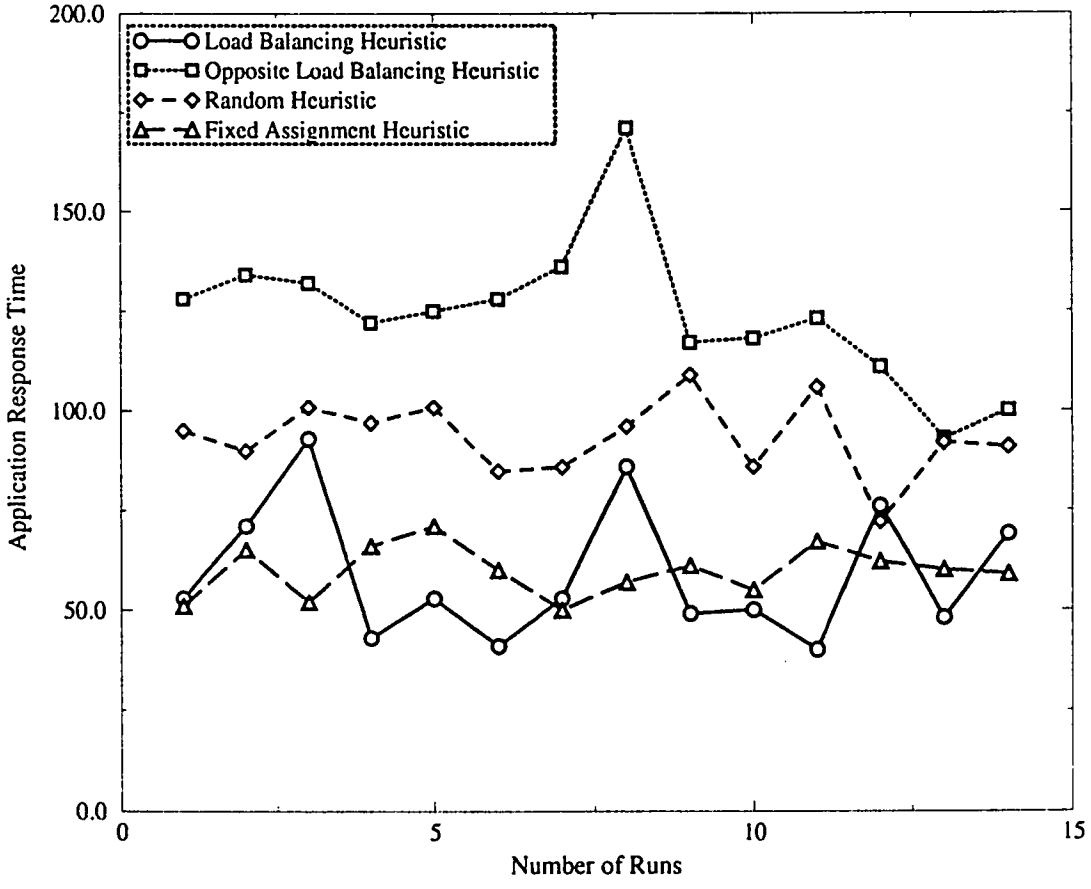


Figure 5.10: **Test3:** Application Completion Time Comparison

5.6.4 Test 4

In this test, the application task graph (Figure 5.11) contains tasks connected in the form of a 4x4 mesh. It contains seven *IO*, six *COM* and three *CPU* type tasks. The granularity range in this case is between five and twenty. The mean granularity is 12.44 whereas the standard deviation in the granularities is 3.8.

5.6.4.1 Total Execution Cost Measurement

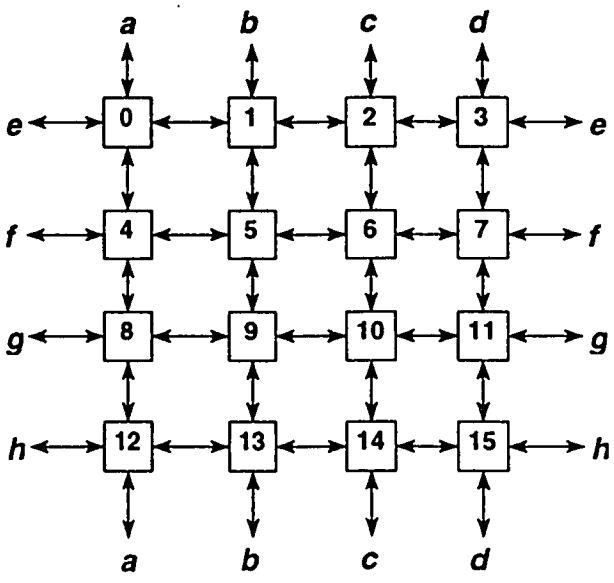
The graph in Figure 5.12 contains the results which are also presented in Table 5.10. Inspite of substantial differences in granularity and more *COM* type of tasks, average of total cost figures show that *lbHeuristic* performs over twice as better as the next best heuristic. The reason for sharp increase in the cost on the six can be attributed to the reasons detailed in Section 5.6.3.1.

Heuristic	Completion Time
LB	1018
OPPLB	2448
RANDOM	1685
Fixed	1478

Table 5.10: **Test4:** Average of Total Execution Cost

5.6.4.2 Average Execution Time Measurement

The graph of Figure 5.13 shows the results which follow expected pattern of higher performance for the *lbHeuristic* case. Further, tests of different types and granularity are expected to follow the same trends as the tree connected application tests.



Task	Type	Granularity
t0	IO	16
t1	IO	6
t2	IO	19
t3	COM	10
t4	COM	9
t5	IO	13
t6	IO	12
t7	COM	17
t8	CPU	16
t9	CPU	9
t10	IO	11
t11	COM	8
t12	CPU	12
t13	COM	18
t14	COM	9
t15	IO	14

Figure 5.11: **Test4:** Task graph and Specification

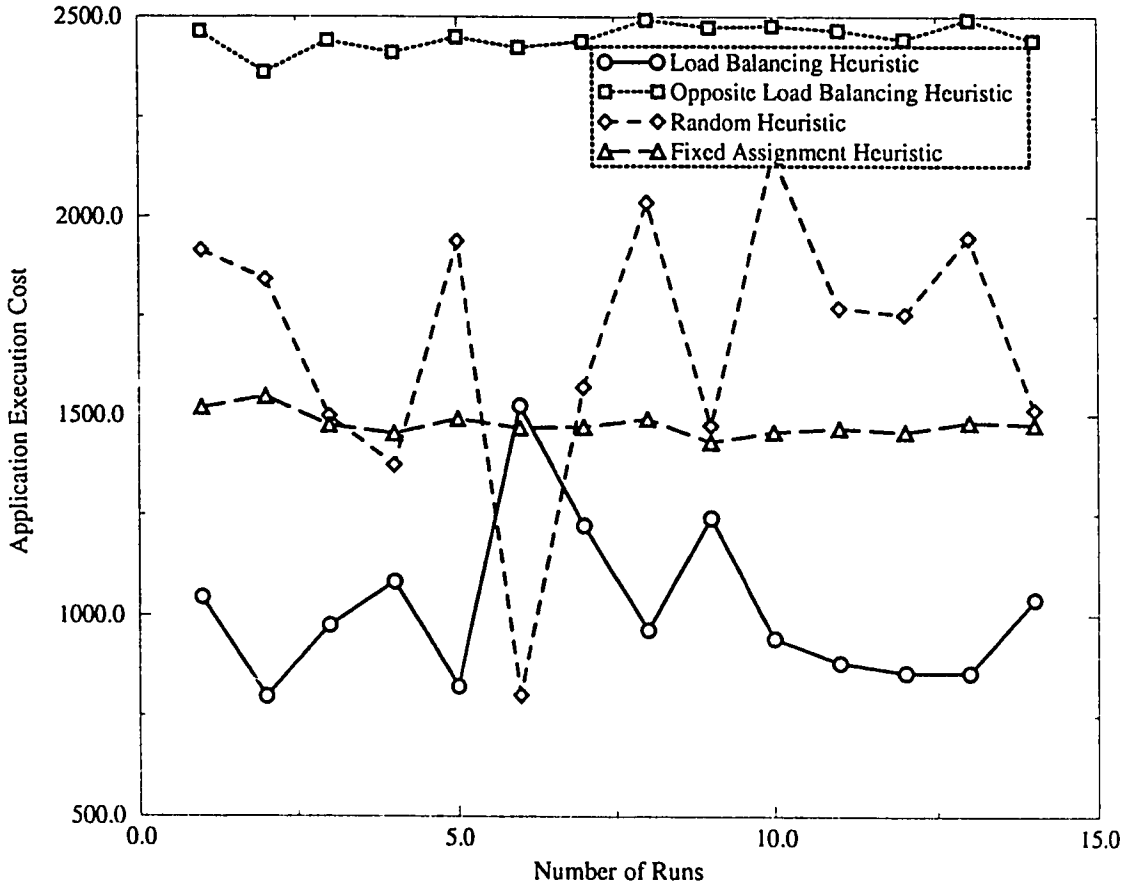


Figure 5.12: **Test4:** Total Execution Cost Comparison

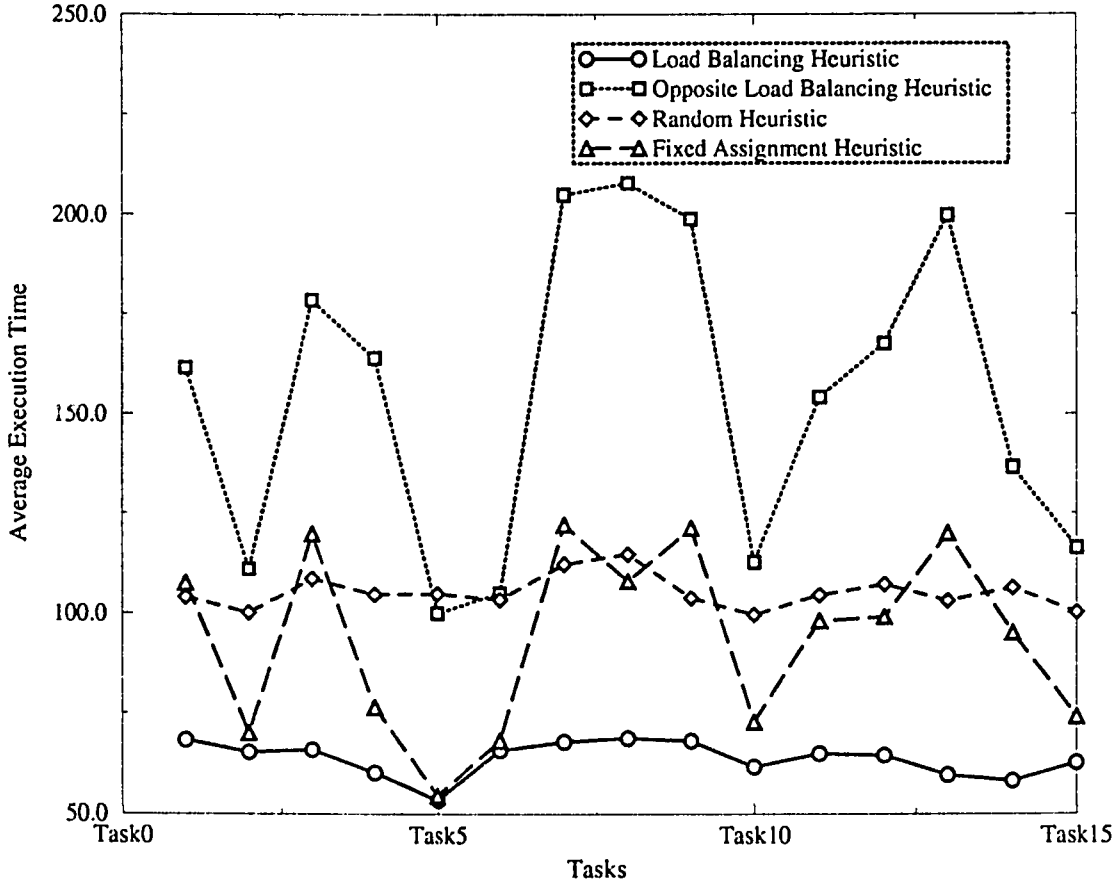


Figure 5.13: **Test4:** Average Task Execution Time Comparison

Chapter 6

Conclusions and Future Work

This thesis addresses the problem of designing and implementing a load balancing framework (*LBFW*) for distributed and parallel applications. The requirements specifications are identified and a detailed design is documented. An implementation is carried out in the *C* language, under *PVM* distributed programming support environment. An interface specification between the framework and user programs is developed. *LBFW* provides user applications with library calls to allow them to interact with it. The load balancing is transparent to the user and is based on a *history* maintenance mechanism. The mapping is guided by applications' accumulated run time characteristics.

For the purpose of experimenting with *LBFW*, a *generic* application test-bed was created. With *generic* applications, it is possible to model distributed or parallel algorithms.

A load balancing heuristic (*lbHeuristic*) takes application and system (network of workstations) characteristics with run history into account. The results from *lbHeuristic* are compared with three other heuristics: random, fixed assignment

and one opposite-to-*lbHeuristic*. The proposed load balancing algorithm performed better in all cases. Performance is measured in terms of completion time of the application, completion cost and average response time per task.

The following is a short list of possible directions for the future enhancements.

- *Comprehensive performance analysis of the developed framework:* The performance analysis can be extended to include some real applications. The performance of *LBFW* can be verified by a mathematical model, which would strengthen its theoretical basis.

- *Tuning load balancing heuristics for different kinds of applications:* Example of this are load balancing algorithms for parallel genetic algorithms, simulation and modelling, distributed ray tracing etc. Application characteristics can influence the mapping in different ways.

- *Distributed LBFW:* In the current implementation, there is one application controller used to make the mapping decision. In order to make *LBFW* highly scalable, an alternative approach is to let each application agent make the mapping decision locally, at the expense of higher cooperation overhead. This is possible if the communication is not a bottleneck.

- *Incorporating fault tolerance in LBFW:* This could involve tolerating faults in any of its agents, such as the application agent, load agent, application controller and the load controller.

- *Supporting LBFW by a dynamic process migration subsystem:* Process migration requires record of low level run time task state, most of which is embed-

ded within the kernel of the operating system. The current implementation of *LBFW* does not interfere with the internals of the kernel. However, a process migration subsystem, if incorporated into *LBFW*, would allow adapting to dynamic load changes throughout the *DCS*.

Appendix A

LBFW Message Description

Table A.1: Description of All *LBFW* messages

Name of the Message	Sender of the Mesg	Receiver of the Mesg	Contents
READTASKFILE	User Agent	Application Controller	UserAgentTid (int), Taskfile Name (char *)
ACKREADTASKFILE	Application Controller	User Agent	-
ENDAPPLICATION	User Agent	Application Controller	UserAgentTid (int), Application Name (char *)
LOADREQUEST	Any Task	Application Controller	UserAgentTid (int)
LOADREPLY	Application Controller	Asking Task	No. of Values (int), Load Vector (double *)
EXECUTETASK	Any Task	Application Controller	UserAgentTid (int), Taskname (char *), AppName (char *), No. of instances (int), argCount (int), argVector (char **)

Table A.1: (Continued)

Message Name	Sender	Receiver	Contents
EXECUTETASK	Application Controller	Application Agent	UserAgentTid (int), Taskname (char *), AppName (char *), No. of instances (int), argCount (int), argVector (char **)
RESULTS	Application Agent	Application Controller	Tid of the reporting App. Agent (int), Taskname (char *), Appname (char *), Result Buffer (char *)
ANSWER	Application Controller	User Task	Buffer got by RESULT message (char *)
FINISHED	Application Agent	Application Controller	Id of terminated task (int), Taskname (char *), Appname (char *), Execution time (long), Number of Tasks communicated (int), Communication Volume (long *)
CURRENTLOAD	Load Controller	Application Controller	Load Vector Size (int), Load Vector (double *)
GETLOAD	Application Controller	Load Controller	
LOADSTATUS	Load Agents	Load Controller	Load Balancing Information data structure (double *)
GIVELOAD	Load Controller	Load Agents	-
TASKTERMINATED	Pvm Daemon	Application Agent	Tid of the terminated task. (pvm_notify message)

Table A.1: (Continued)

Message Name	Sender	Receiver	Contents
TASKRESULT	User Task	Application Agent	Its Tid(int), Result Buffer (char *)
ABORT	User Agent	Application Controller	-
HALT	Application Controller	Application Agents	-
TIDLIST	Application Agent	Application Controller	Number of tasks (int), Task identifier list (int *)
TIDLIST	Application Controller	User Agent	Number of tasks (int), Task identifier list (int *)

Appendix B

LBFW User Guide

This user's guide to *LBFW* contains additional information to a programmer intending to use it. *LBFW* is based on the message passing harness *PVM* [18]. It provides a wrap over the functionality of *PVM*. It provides routines having similar syntax and semantics as those of *PVM*. The following presentation assumes that the application programmer is familiar with the nuances of programming under *PVM*.

B.1 Compiling LBFW

Since *LBFW* is an application on top of *PVM*, it is assumed that *PVM* is already installed on the target network. For compiling *LBFW*, copy the *lbfw.tar* file to the the *PVM* home directory. Un-tar it using *tar xvf lbfw.tar* command on the command line. One of the directories created is *lbfwFramework* in the *PVM_ROOT* directory. The file *Makefile.aimk* contains the makefile for *LBFW*. For making *LBFW*, just type *aimk*. This creates the following executable in the *PVM_ROOT/bin/PVM_ARCH* directory:

- appController: This contains the application controller.
- appAgent: This is the application agent.
- centralLoad: This is the load controller and
- distLoad: This is the load agent.

It also generates, the application programmer's interface library for *LBFW* in *PVM_ROOT/lib/PVM_ARCH* by the name *liblbfw.a*. It has to be made sure that *PVM* and *LBFW* are compiled manually for all the target architectures intended to be used as platforms for application programming.

B.2 Compiling LBFW based applications

The examples programs for *LBFW* are provided in the *lbfwExamples* directory under *PVM_ROOT* directory. A C program that makes *LBFW* library calls needs to be linked with *liblbfw.a* library. This is already customized in a custom makefile.

B.3 Library routines for LBFW application programming

- `executeTask (taskName, appName, nInstances, argVect, tids, returnFlag)`
  This routine starts up *nInstances* copies of an executable file *taskName* on the *VM*. The location of execution of the task is decided by *LBFW*. The argument *appName* identifies the user perceived application name. Both of the previous parameters are assumed to be non NULL quantities. *argVect* specifies a NULL terminated pointer to an array of arguments to *taskName*. If there are no command line arguments then *argVect* is NULL. The *returnFlag* specifies whether the task id has to be returned or not. If *returnFlag* is *YES* then, the task ids of the executed tasks are returned in the vector *tids*; if some of the tasks could not be started the corresponding error codes are placed in it.

- `int giveApplicationControllerTid ()`
  This utility routine may be used by an advanced programmer to interact directly with the application controller. This routine returns the task id of the application controller. If it could not be found, then this returns a value of -1.

- `int giveLoadInfo (LoadVector)`
  This utility is very useful for the case of *Data Parallel* applications. Typically, such applications are based on *agenda parallelism* where a supervisor task partitions the data and based on the current load information, data is partitioned between a number of workers. This routine expects as argument a pointer to a real array *LoadVector*, which can contain atleast *N* values (where *N* is the number of hosts in the *VM*). It then returns the current normalized load values, as seen by the application controller, at that particular instant of time.

- `int giveHostCount ()`
  This utility is again targeted in helping application programs in data partitioning based on the number of hosts in the *VM*. This routine returns the number of hosts in the current *VM* setup, which could then be used for partitioning the data.

- `int absolutePathToFilename (absolutePath, filename)`
  The need of such a utility is frequent in the case, where the task name itself needs to be known for making some decisions. By default, accessing the *argv*[0] variable will not solve the problem as *PVM* uses absolute path names for executing tasks. This routine takes the absolute path name as its first argument and returns its last name as specified by the user in the taskfile.

- `int lbpvm_startApplication (userAgentTid, taskFileName)`
  This routine registers a user application to *LBFW*. The first argument is the task id of the user agent and the second argument specifies a non-NULL taskfile name. This routine sets up the in-memory data structures for the requested application in the application controller. This routine *must* be called once before any call to the routine *executeTask* can be made. This is because, *executeTask* invokes the load balancing module, which relies on the history information for making load balancing decisions. Even if the application is executed for the first time, this data structure is maintained with all the statistics initialized to -1.

- `int lbpvm_endApplication(appName)`
  This routine is called by the user agent, when the application finishes its execution. It initiates the saving of the history information associated with the application. This is usually the last statement of the user agent.

- `lbpvm_initCommunication(appName, taskName)`
  LBFW transparently maintains the communication statistics between all the communicating tasks of an application. This routine is called *once* in every task, if it has to communicate with some other task. This sets up the *applicationName* and the *taskName* of the tasks in the internal buffers, which are sent automatically to any task which receives any data from *taskName*.

- `int lbpvm_initsend (encoding)`
  This routine is a direct enhancement of the *PVM pvm_initsend* routine. This does the job of initializing the send buffers, as in the case of *PVM*, and in addition it initializes the byte count to zero to start a new communication. This initializing procedure is used with non-blocking send routines.

- `int lbpvm_initSyncSend (encoding)`
  This routine provides a blocking send. This is useful when the sender needs to block, making sure that the receiver receives the message which has been sent. In the background, this does a handshake with the receiver.

- `int lbpvm_pk* ()`
  These routines are direct extension of *pvm_pk** routines having the same syntax and the semantics. The only difference being that they help in maintaining

statistics about the volume of communication in terms of bytes transferred between communicating tasks.

- `int lbpvm_upk* ()`
  These routines are direct extension of *pvm_upk** routines having the same syntax and the semantics. The only difference being that they help in maintaining statistics about the volume of communication in terms of bytes transferred between communicating tasks.

- `int lbpvm_send(destinationTid, messageTag)`
  This routine is an extension of the *pvm_send* routine. Its syntax and semantics are the same as the former. The only additional function is that it attaches an *LBFW Header* to the message about the sender so that the receiver knows who the sender is. This is used by the corresponding receive routine to maintain communication volume information.

- `int lbpvm_SyncSend(destinationTid, messageTag)`
  This routine is applicable with *lbpvm_initSyncSend* routine. This internally does the hand-shaking with the receiver and returns only after the handshake is successful.

- `int lbpvm_mcast(tidList, nTasks, messageTag)`
  This routine again is a direct extension of *pvm_mcast* but is needed for maintaining the communication volume information. It has the same semantics as the former.

- `int lbpvm_StartRecv(sourceTid, messageTag)`
  Any communication, from the receiver's point of view in the context of *LBFW* is bracketed between a call to this and the *lbpvm_EndRecv* routine. This routine sets up suitable variables for counting the byte count between any two communicating tasks on a session basis. Thus each communication session has to be as specified. The syntax and semantics of this routine are the same as that of the *pvm_recv* routine.

- `int lbpvm_StartnRecv(sourceTid, messageTag)`
  This routine is *LBFW's* equivalent of the non-blocking receive provided by *PVM*. The motivation for this is same as in the previous routine. Its syntax and semantics are the same as pvm_nrecv routine.

- `int lbpvm_EndRecv()`
  This routine indicates the termination of a *logical* session of communication between a sender and a receiver. This routine stores the values in the buffer to the *task execution database* of the application agent. This routine is intelligent

enough to find out if the sender is expecting an acknowledgement. If so it sends the suitable acknowledgement.

Bibliography

[1] Vivek Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[2] Dan I. Moldovan, *Parallel Processing From Applications to Systems*, vol. 1, Morgaan Kaufmann, San Mateo, CAA 94403, 1 edition, 1993.

[3] Virginia Mary Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems", *IEEE Transactions on Computers,*, vol. 37, no. 11, pp. 1384–1397, November 1988.

[4] Nicholas S. Bowen, Christos N. Nikolaou, and Arif Ghafoor, "On the Assignment Problem of Arbitrary process systems to Heterogeneous Distributed Computer Systems", *IEEE Transactions on Computers*, vol. 41, no. 3, pp. 257–273, March 1992.

[5] M. Bozyigit and M. Melhi, "A Load Balancing Framework for Distributed Systems", *International Journal of Computer Systems Science and Engineering*, 1996, In Publication.

[6] A.J Harget and I.D. Johnson, *Load Balancing algorithms in loosely coupled distributed systems : a survey*, chapter 6, Butterworths, 1990.

[7] H.S.M Zedan, Ed., *Distributed Computer Systems*, Butterworths, 1990.

[8] Domenico Ferrari and S. Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications", Tech. Rep., University of California, Berkely, 1987.

[9] Thomas Kunz, "The Influence of Different Workload Descriptors on a Heuristic Load Balancing Scheme", *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725–730, July 1991.

[10] Matt W. Mukta, "Estimating Capacity for Sharing in a Privately Owned Workstation Environment", *IEEE Transactions on Software Engineering*, vol. 18, no. 4, pp. 319–328, April 1992.

[11] David M. Ogle, Karsten Schwan, and Richard Snodgrass, "Application Dependent Dynamic Monitoring of Distributed and Parallel Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 762–778, July 1993.

[12] Kumar K. Goswami, Murthy Devarkonda, and Ravishankar K. Iyer, "Prediction Based Dynamic Load Sharing Heuristics", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 638–648, June 1993.

[13] Anna Hac and Xiaowei Jin, "Dynamic Load Balancing in a Distributed system Using Sender-Initiated Algorithm", *Journal of Systems and Software*, vol. 11, no. 1, pp. 79–94, January 1990.

[14] Anna Hac and Xiaowei Jin, "A Decentralized Algorithm for Dynamic Load Balancing with File Transfer", *Journal of Systems and Software*, vol. 16, no. 6, pp. 37–52, June 1991.

[15] Mikhail J. Atallah, Christina Lock Black, and Dan C. Marinescu, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations", *Journal of Parallel and Distributed Computing*, vol. 16, no. 1, pp. 319–327, 1992.

[16] Kam Hong Shum and Muslim Bozyigit, "A load Distribution through Competition for Workstation Clusters", Tech. Rep., University of Cambridge, Cambridge, 1993.

[17] Brian K. Schmidt and Vaidy S. Sunderam, "Empirical Analysis of Overheads in Cluster Environments", *Concurrency: Practice and Experience*, vol. 6, no. 1, pp. 1–32, February 1994.

[18] A. L. Beguelin, J. J. Dongarra, G. A. Geist, W. C. Jiang, R. J. Manchek, B. K. Moore, and V. S. Sunderam, "Parallel Virtual Machine System 3.2", Tech. Rep., PVM consortium, 1992.

[19] John B. Weissman and Andrew S. Grimshaw, "A Framework for Partitioning Parallel Computations in Heterogeneous Environments", *Concurrency: Practice and Experience*, vol. 7, no. 5, pp. 455–478, August 1995.

[20] Gregory D. Peterson and Roger D. Chamberlain, "Sharing Networked Workstations: A Performance Model", *Proceedings of 6th IEEE Symposium on Parallel and distributed Processing, Dallas Texas*, vol. 1, no. 1, pp. 308–315, October 26-29 1994.

[21] S. N. Haq, M. Bozyigit, S. Ghanta, and S. K. Naseer, "Design of a Load Balancing Framework for Distributed and Parallel Applications", *International Conference on Parallel and Distributed Techniques and Applications (PDPTA-95), Athens-Georgia*, vol. 1, no. 1, pp. 979–993, November 3-4 1995.

[22] M. D. Feng and C. K. Yeun, "Dynamic Load Dalancing on a Distributed System", *Proceedings of 6th IEEE Symposium on Parallel and distributed Processing, Dallas Texas*, vol. 1, no. 1, pp. 318–325, October 26-29 1994.

[23] Marc H. Willebeek-LeMair and Anthony Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979–993, September 1993.

[24] D. P. Bertskas and J. N. Tsitsiklis, Eds., *Parallel and Distributed Computation: Numerical Methods.*, Prentice Hall: Englewood Cliffs, 1989.

[25] Frank C. H. Lin and Robert Keller, "The Gradient Model of Load Balancing Method", *IEEE Transactions on software Engineering*, vol. SE-13, no. 1, pp. 32–38, January 1987.

[26] J. Worlton, *Toward a Science of Parallel Computation*, Research Monographs in Parallel and Distributed Computing. New York, 1986.

[27] Syed Khwaja Naseer, "A Process Migration Subsystem for Distributed Applications", Master's thesis, Department of Computer Science, KFUPM, 1995.

[28] Massimo Bernaschi and Giorgio Richelli, "Development and Results of PVMe on IBM 9076 SP1", *Journal of Parallel and Distributed Computing*, vol. 29, no. 1, pp. 75–83, 1995.

[29] D. P. Bertskas and R. Gallager, *Computer Communication Networks.*, Prentice Hall: Englewood Cliffs, 1989.

[30] Selim G. Akl, *The Design and Analysis of Parallel Algorithms*, vol. I of *Prentice Hall Computer Science*, Prentice Hall, Englewood Cliffs NJ, USA., 1990 edition, January 1989.

[31] Marc J. Rochkind, *Advanced Unix Programming*, Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

Vita

- SYED NISAR UL HAQ
- Born in Hyderabad, India on January 6, 1971
- Received Bachelor's degree in Computer Science and Engineering from Osmania University Hyderabad, India in July , 1992.
- Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in January, 1996.